User Documentation for IDA, a Differential-Algebraic Equation Solver for Sequential and Parallel Computers

Alan C. Hindmarsh
Lawrence Livermore National Laboratory

Allan G. Taylor
Lawrence Livermore National Laboratory

Center for Applied Scientific Computing

UCRL-MA-136910
December 1999

USER DOCUMENTATION FOR IDA, A DIFFERENTIAL-ALGEBRAIC EQUATION SOLVER FOR SEQUENTIAL AND PARALLEL COMPUTERS*

ALAN C. HINDMARSH AND ALLAN G. TAYLOR†

1. Introduction. IDA is a general purpose solver for the initial value problem for systems of differential-algebraic equations (DAEs). The name IDA stands for Implicit Differential-Algebraic solver. IDA is based on DASPK [4, 5], but is written in ANSI-standard C rather than Fortran 77. Its most notable feature is that, in the solution of the underlying nonlinear system at each time step, it offers a choice of Newton/direct methods or an Inexact Newton/Krylov (iterative) method. Thus IDA shares significant modules previously written within CASC at LLNL to support the ordinary differential equation (ODE) solvers CVODE [9, 10] and PVODE [7, 8], and also the nonlinear system solver KINSOL [13].

The Newton/Krylov method uses the GMRES (Generalized Minimal RESidual) linear iterative method [12], and requires almost no matrix storage for solving the Newton equations as compared to direct methods. However, the GMRES algorithm allows for a user-supplied preconditioner matrix, and for most problems preconditioning is essential for an efficient solution.

The IDA package has been arranged so that selecting one of two forms of a single module in the compilation process will allow the entire package to be created in either sequential (serial) or parallel form. The parallel version of IDA uses MPI (Message-Passing Interface) [11] and an appropriately revised version of the vector module `NVECTOR`, to achieve parallelism and portability. The parallel form of IDA is intended for a SPMD (Single Program Multiple Data) programming model with distributed memory, in which all vectors are identically distributed across processors. In particular, the vector module `NVECTOR` is designed to help the user assign a contiguous segment of a given vector to each of the processors for parallel computation. In implementing IDA, several primitives were added to `NVECTOR` beyond those originally written for PVODE and KINSOL.

IDA was developed and tested on a cluster of SUN-SPARC workstations. It is currently being used to solve radiation-diffusion transport systems of up to 663 million unknowns on an IBM SP multiprocessor (1024 CPUs).

The remainder of this document is organized as follows: Section 2 sets the mathematical notation and summarizes the basic methods, and Section 3 summarizes the organization of the IDA solver. Section 4 provides complete usage instructions, and Section 5 gives the interface between IDA and possible additional linear solver modules. Section 6 describes a preconditioner module, and Section 7 describes a set of example problems. Finally, Section 8 discusses the availability of IDA.

2. Mathematical Considerations. The IDA code is a C implementation of a previous code, DASPK, a DAE system solver written in Fortran by Petzold, Brown, and Hindmarsh

* Research performed under the auspices of the U.S. Department of Energy, by Lawrence Livermore National Laboratory under contract W-7405-ENG-48.

† Center for Applied Scientific Computing, L-561, LLNL, Livermore, CA 94551.

[4, 1]. IDA solves the initial value problem for a DAE system of the general form

$$F(t, y, y') = 0 \ , \tag{1}$$

where y and F are vectors in $\mathbf{R}^N$, t is the independent variable, and initial conditions $y(t_0) = y_0, y'(t_0) = y'_0$ are given. (Often t is time, but it certainly need not be.)

Unlike the situation for ODE systems, the initial vectors y_0 and y'_0 are not arbitrary, but must be consistent with the system (1). For a class of problems that includes so-called semi-explicit index-one systems, IDA includes a routine that computes consistent initial conditions from a user's initial guesses [5]. For this, the user must identify subvectors of y (not necessarily contiguous), denoted y_d and y_a, which are its differential and algebraic parts, respectively, such that F depends on y'_d but not on any components of y'_a. The assumption that the system is "index-one" means that for a given t and y_d, the system $F(t, y, y') = 0$ defines y_a uniquely. In this case, a solver within IDA computes y_a and y'_d at $t = t_0$, given y_d and an initial guess for y_a. A second available option with this solver also computes all of $y(t_0)$ given $y'(t_0)$; this is intended mainly for quasi-steady state problems, where $y'(t_0) = 0$ is given. For problems that do not fall into either of these categories, the user is responsible for passing consistent values, or risk failure in the numerical integration.

IDA integrates the system (1) with Backward Differentiation Formula (BDF) methods, implemented in a variable-order, variable-step form. The method orders range from 1 to 5, and the BDF of order k is given by the multistep formula

$$\sum_{i=0}^{k} \alpha_{n,i} y_{n-i} = h_n y'_n \ , \tag{2}$$

where y_n and y'_n are the computed approximations to $y(t_n)$ and $y'(t_n)$, respectively, and the stepsize is $h_n = t_n - t_{n-1}$. The coefficients $\alpha_{n,i}$ are uniquely determined by the order k, and the history of the stepsize. The application of the BDF (2) to the DAE system (1) results in a nonlinear algebraic system to be solved at each step:

$$G(y_n) \equiv F(t_n, y_n, h_n^{-1} \sum_{i=0}^{k} \alpha_{n,i} y_{n-i}) = 0 \ . \tag{3}$$

Regardless of the method options, the solution of the nonlinear system (3) is accomplished with some form of Newton iteration. This leads to a linear system for each Newton correction, of the form

$$J[y_{n(m+1)} - y_{n(m)}] = -G(y_{n(m)}) \ , \tag{4}$$

where $y_{n(m)}$ is the mth approximation to y_n. Here J is some approximation to the system Jacobian

$$J = \frac{\partial G}{\partial y} = \frac{\partial F}{\partial y} + \alpha \frac{\partial F}{\partial y'} \ , \tag{5}$$

where $\alpha = \alpha_{n,0}/h_n$. The scalar α changes whenever the stepsize or method order changes.

During the course of integrating the system, IDA computes an estimate E_n of the local truncation error at the nth time step, and requires this to satisfy the inequality

$$\|E_n\|_{wrms} < 1 \ . \tag{6}$$

This test imposes tolerances on the local errors by way of the weighted root-mean-square norm, which is defined by

$$\|E_n\|_{wrms} = \left[\frac{1}{N}\sum_{i=1}^{N}(E_n^i/w^i)^2\right]^{1/2} \ .$$

Here a superscript i denotes the ith component, and the ith weight is

$$w^i = rtol|y^i| + atol^i \quad \text{or} \quad w^i = rtol|y^i| + atol \ . \tag{7}$$

This permits an arbitrary combination of relative and absolute error control. The user specifies a scalar relative error tolerance *rtol* and an absolute error tolerance *atol* which may be either an N-vector or a scalar (as indicated in (7) above). Since these tolerances define the allowed error per step, they should be chosen conservatively.

IDA varies both the stepsize h_n and the order k in an attempt to produce a solution with the minimum number of steps, but always subject to the local error test (6). After a step at order k, the local truncation errors at orders $k-1$ and (often) $k+1$ are also estimated, and a change of order is considered on the basis of the three error norms. See [1] for details.

Normally, IDA takes steps until a user-defined output value $t = tout$ is overtaken, and then computes $y(tout)$ by interpolation. However, a "one-step" mode option is available, where control returns to the calling program after each step. There are also options to force IDA not to integrate past a given stopping point $t = t_{stop}$.

For the solution of the linear systems (4), IDA includes both direct and iterative methods. In the direct case, the Jacobian J defined in (5) can be treated as either dense or banded, and in each case, the user can either supply an approximation to J or have IDA compute one internally by difference quotients.

At present, the only iterative method included in IDA is the Scaled Preconditioned GMRES method, denoted SPGMR. Writing the linear system (4) abstractly as $Ax = b$, we seek a preconditioner matrix P that approximates A, but for which linear systems $Px = b$ can be solved easily. Preconditioning is applied on the left only, giving the equivalent system $(P^{-1}A)x = P^{-1}b$. Scaling is included explicitly in the SPGMR algorithm, using a diagonal scaling matrix D whose diagonal elements are the weights w^i of (7). Thus the system actually solved with the GMRES method is

$$(D^{-1}P^{-1}AD)(D^{-1}x) = D^{-1}P^{-1}b \ , \quad \text{or} \quad \bar{A}\bar{x} = \bar{b} \ . \tag{8}$$

From an initial guess $\bar{x}_0$, an approximate solution $\bar{x}_m = \bar{x}_0 + z$ is obtained for $m = 1, 2, \ldots$ (until convergence), with z chosen from the Krylov subspace $K_m = span\{r_0, \bar{A}r_0, \ldots, \bar{A}^{m-1}r_0\}$ of dimension m, where r_0 is the initial (transformed) residual $\bar{b} - \bar{A}\bar{x}_0$. Each Krylov iteration requires one matrix-vector multiply operation $\bar{A}v$, which is a combination of scalings

and multiplications by A and by P^{-1}. Multiplication of a given vector v by A requires the product Jv, and that is approximated by a difference quotient $[G(y+\sigma v) - G(y)]/\sigma$ with a suitably small σ. Multiplication by P^{-1} is to be provided by the user, and is generally problem-dependent. Details of the SPGMR algorithm in combination with the BDF integration method used here can be found in [4].

IDA provides options to impose inequality constraints on the solution—a feature that expands considerably on that in DASPK. By way of an input vector of flags, the user of IDA can specify that each component of y is to be positive, non-negative, non-positive, or negative. These constraints are applied at each time step, and also during the optional calculation of consistent initial conditions.

3. Code Organization. One can visualize IDA as being organized in layers, as shown in Fig. 1. Viewed this way, the user's program is at the top level. This program, with associated user-supplied routines, makes various initialization calls, manages input/output, and calls the `IDA` main module for the solution of the problem.

At the next level down, the `IDA` main module controls the solution of the DAE initial value problem, including initial condition calculation, implicit stepwise integration, and associated Newton iterations. By design, this module is independent of the choice of linear system method being used. The three principal user-callable routines are: `IDAMalloc`, for memory allocation and basic initializations; `IDACalcIC`, for consistent initial condition calculation; and `IDASolve`, for integration of the DAE system. `IDA` calls the user-supplied routine `res` defining F, and accesses the user-selected linear system solver.

At the third level are the linear system solvers, which at present are `IDADENSE`, `IDABAND`, and `IDASPGMR`. Each of these provides an interface to a corresponding generic solver for linear systems by a dense, banded, or SPGMR algorithm. The direct method modules access the user's Jacobian routine `jac` if one is supplied. The `IDASPGMR` module accesses the user-supplied preconditioner solve routine `psolve`, and possibly also a user-supplied routine `precond` that computes and preprocesses the preconditioner. As a companion to the `IDASPGMR` module, the IDA package includes a module called `IDABBDPRE`, which builds a band-block-diagonal preconditioner for use with SPGMR; see Section 6 for full details.

Each linear solver module interfaces to the `IDA` module through five functions, each having a fixed call list. These functions and their purposes are as follows:

- `linit`: initialize and allocate memory specific to the linear solver;
- `lsetup`: evaluate and preprocess the Jacobian or preconditioner;
- `lsolve`: solve the linear system;
- `lperf`: monitor performance and issue warnings;
- `lfree`: free memory.

The connections between `IDA` and these functions are set at link time. As a result of this organization, the `IDA` module is independent of the linear solver, and the set of linear solvers is expandable. See Section 5 and Ref. [10] for further details on this design.

Three supporting modules reside at the fourth level: `LLNLTYPS`, `LLNLMATH`, and `NVECTOR`. The first of these defines types `real`, `integer`, and `boole`. The second specifies several basic mathematical functions such as power functions and unit roundoff. The third is discussed further below.

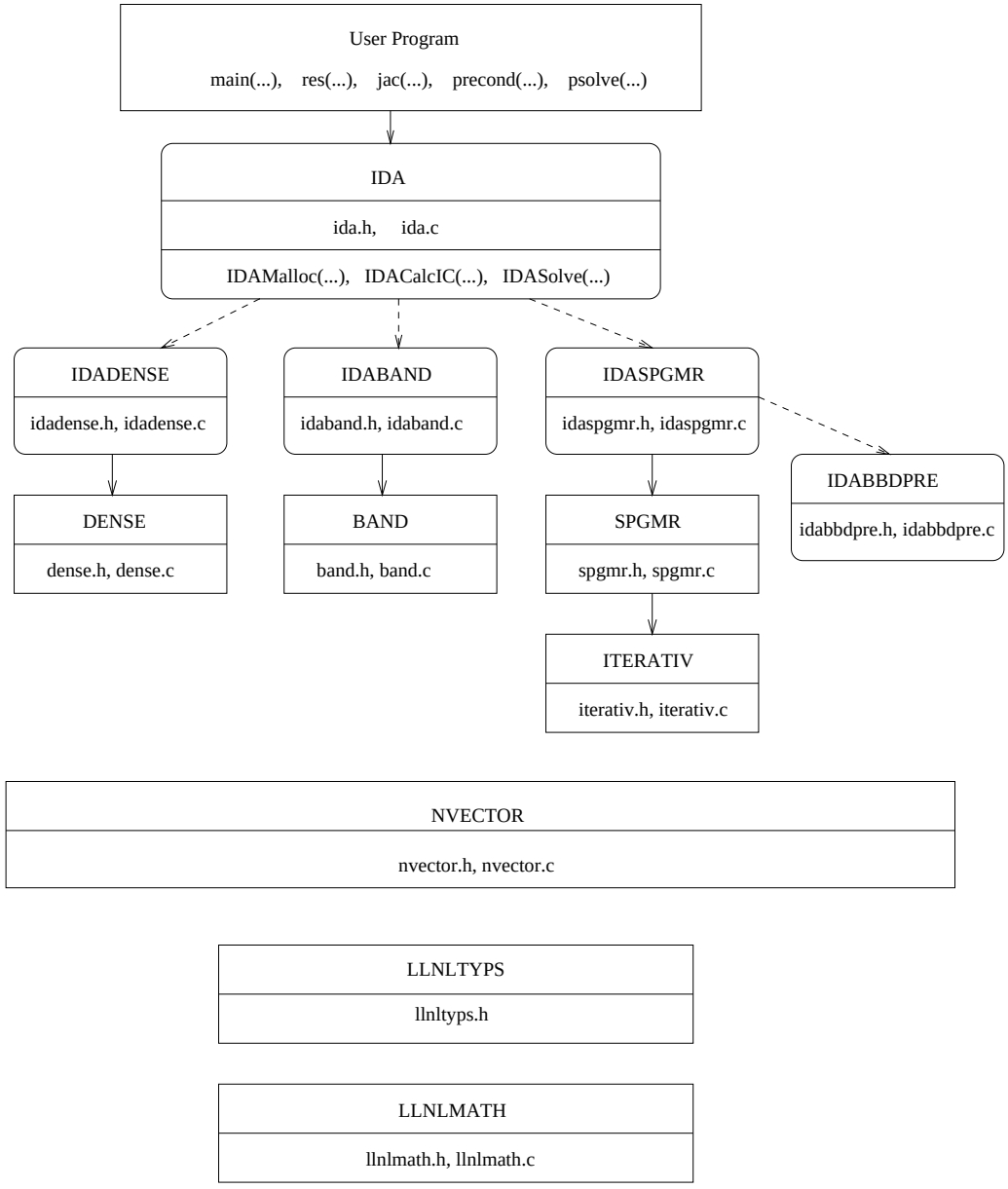

FIG. 1. *Overall structure of the IDA package. Modules comprising the central solver are distinguished by rounded boxes, while the user program, generic linear solvers, and generic auxiliary modules are in unrounded boxes.*

The key to being able to move from the sequential computing environment to the parallel computing environment lies in the NVECTOR module. This contains a set of mathematical operations on N-vectors, and is shared with the solvers CVODE/PVODE and KINSOL. The operations handled in this module are vector linear combinations, scaling, vector copy, vector norms, scalar products, and so forth. By separating these operations from the rest of the code, all operations in IDA with significant potential for parallel computation have been isolated. Then two different sets of kernels, both with the same routine names and a common interface, allow parallel computation to be implemented in a clean manner in these codes. The solvers access the kernels without referring directly to the underlying vector structure. This is made possible by using an abstract data type, `type N_Vector`, for all N-vectors, and another data type, `type machEnvType`, that describes a block of the machine environment data. The latter block is empty in the serial case. The parallel version of the module includes a function to define the block of type `machEnvType`, and another to free that block.

In the parallel version of IDA, all N-vectors are to be distributed over the various processors in the same way, so that, in a sense, each processor is solving a contiguous subset of the DAE system. For any given vector operation, each processor performs the operation on its contiguous elements of the input vectors, of length (say) `Nlocal`, followed by a global reduction operation where needed. In this way, vector calculations can be performed simultaneously with each processor working on its block of the vector. IDA uses the MPI (Message Passing Interface) system [11] for all inter-processor communication. This achieves a high degree of portability, since MPI is becoming widely accepted as a standard for message passing software.

Because the IDA interface to the vector kernels is independent of the vector structure, a user could supply their own version of these kernels to better fit their application data structures, or to accommodate a different parallel machine environment. All references to parallelism are in the kernel, and thus the user would handle all parallel aspects in this case.

The algorithms used in DASPK, as modified for IDA, have several unique features not present in KINSOL or PVODE, notably the way that constraints are handled. As a result, three new vector kernels were written and added to the NVECTOR module in support of IDA, namely `N_VOneMask`, `N_VConstrMask`, and `N_VMinQuotient`. These additions to the 'common' version of NVECTOR are completely transparent to CVODE/PVODE and KINSOL.

The IDA solver keeps its various work spaces and status data in a memory block, whose contents are not seen by the user. However, a pointer to this block, once created, is returned to the user and must be passed back in subsequent calls to the various solver routines. In this way, the internal memory for the IDA solution of a specific problem is retained in a safe manner. A similar memory block is retained by each generic linear system solver in the package.

The coding style and structure of IDA was based on both the style and structure of the pre-existing CVODE/PVODE and KINSOL codes. This was predicated upon the requirement that the same vector kernel implementation and the same generic dense, band, and GMRES solvers be used in all these codes. The original DASPK routines were completely restructured to eliminate awkward coding constructs. Considerable simplification and clarification of the internal calling sequences resulted from this process. Of course, the

resulting C language structure maintains relative privacy for definitions for each portion of the code. The resulting code has proven to be readily adaptable to either sequential or parallel execution.

The modules in the IDA solver package, along with some specifics on their content, are shown in Table 1. For each module there is a source file and a header file (except for `LLNLTYPS`, which has only a header file).

Module name	User-callable routines	other contents
`IDA`	`IDAMalloc`, `IDACalcIC`, `IDASolve`, `IDAFree`	system function type `ResFn`; linear solver function pointers
`IDADENSE`	`IDADense`	`IDASpgmrDenseJacFn` type
`IDABAND`	`IDABand`	`IDASpgmrBandJacFn` type
`IDASPGMR`	`IDASpgmr`	`IDASpgmrPrecondFn` type `IDASpgmrPrecondSolveFn` type
`SPGMR`		`SpgmrMalloc`, `SpgmrSolve`, `SpgmrFree`
`ITERATIV`		Routines in support of `SPGMR`
`IDABBDPRE`	`IBBDAlloc`	`IDALocalFn` type, `IDACommFn` type
`NVECTOR`	`PVecInitMPI`, `PVecFreeMPI`, 19 other vector kernels	Type `N_Vector`; vector macros `N_VMAKE`, `N_VDATA`, etc.
`LLNLMATH`		`UnitRoundoff`, `RPowerI`, `RPowerR`, `RSqrt`; Macros `MIN, MAX, ABS, SQR`
`LLNLTYPS`		Types `real`, `integer`, `boole`

TABLE 1
Modules in the IDA solver package

4. Using IDA. This section describes the use of IDA. We begin with a brief overview, in the form of a skeleton user program. Following that are detailed descriptions of the interface to the various user-callable routines, and of the user-supplied routines. Finally, there are comments on usage under C++ and on exported IDA data types.

The Appendix displays an example program, called `webpk`, offered as a detailed template to assist users in preparing IDA applications. This example is based on a PDE system from a food web (predator-prey) model, solved with the parallel version of IDA, using the Krylov linear system method. Other examples provided with the package illustrate the serial version and other method options. The serial examples are `robx, heatsb, heatsk,` and `websb`, and the parallel examples are `heatpk, heatbbd, webpk,` and `webbbd`.

4.1. Overview of Usage. The following is a skeleton of the user's main program (or calling program) as an application of IDA. The user program is to have these steps in the order indicated, although some of the steps given are relevant only to a parallel machine environment. For the sake of brevity, we defer many of the details to the later subsections.

1. `#include` header files needed, to obtain various type definitions, enumerations, macros, etc. The files include `llnltyps.h, llnlmath.h, ida.h, nvector.h`, one or more of the files `idadense.h, idaband.h, idaspgmr.h, idabbdpre.h` associated with the linear system solvers, and (in a parallel environment) `mpi.h`.
2. `MPI_Init(&argc, &argv)` to initialize MPI if used by the user's program. Here `argc` and `argv` are the command line argument counter and array received by `main`.
3. `Nlocal =` the local vector length (the sub-vector length for this processor), and `Neq =` the global vector length (the problem size N, and the sum of all the `Nlocal`).
4. `machEnv = PVecInitMPI(comm, Nlocal, Neq, &argc, &argv)` followed by `if (machEnv == NULL) return(1)`, to initialize the `NVECTOR` module. Here `comm` is the MPI communicator, which may be set by suitable MPI calls, for a proper subset of the active processors, or else set to either `NULL` or `MPI_COMM_WORLD`, to specify that all processors are to be used.
5. Set `N_Vector yy` and `N_Vector yp` to initial values for y and y', respectively. For this, use the macro `N_VMAKE(yy, ydata, machEnv)` if an existing array `ydata` contains the initial values of y. Otherwise, make the call `yy = N_VNew(Neq, machEnv)` and load initial values into the array defined by `N_VDATA(yy)`. Create `yp` similarly. Depending on user options, also create the vector `id` of differential/algebraic component flags and/or the vector `constraints` of inequality constraint flags.
6. `idamem = IDAMalloc(Neq, resfn, rdata, t0, yy, yp, itol,...)` followed by `if (idamem == NULL) return(1)`, to allocate and initialize IDA's internal memory, and obtain a pointer to the IDA memory block. Here `rdata` is a pointer to a user-defined data block which is made available to the user residual function `resfn`. This call also includes tolerances `rtol, atol` and a tolerance type flag `itol`.
7. Specify the linear system solver to be used by making one of the calls: `flag = IDADense(...)` or `flag = IDABand(...)` or `flag = IDASpgmr(...)` followed by a test `if (flag != 0) return(1)`.
8. Optionally, correct the initial values in `yy, yp` with the call `flag = IDACalcIC(idamem, icopt, ...)`; `if (flag != 0) return(1)`.
9. `flag = IDASolve(idamem, tout, tstop, &tt, yy, yp, itask)` (within a loop) to perform the integration of the DAE system from the current point to $t =$ `tout`. In this call, set `itask = NORMAL` to integrate past `tout` and interpolate. (Alternatively, set `itask = ONE_STEP` to take one step forward and return.) Following the call, process the vectors `yy` and `yp`, the computed solution y and y' at $t =$ `tt`. Also, test for the condition `flag < 0` to detect a failure.
10. `IDAFree(idamem); PVecFreeMPI(machEnv); MPI_Finalize()` to free the IDA internal memory block and machine-dependent data, and to close MPI.
11. Supply a function `resfn` to define the DAE system residual function $F(t, y, y')$. This function is to have the form `resfn(Neq, tt, yy, yp, res, rdata)`. Likewise supply any routines for Jacobian evaluation or preconditioning, as required by the linear system solver chosen in step 7 above.

As indicated above, error conditions are possible at many of the steps, and are flagged by nonzero return values. In addition, error messages are issued in most cases.

In the case of a sequential (serial) machine environment, the user program is of course considerably simpler. The skeleton program in that case reads as follows. Again, complete details are given later.

1. `#include` header files needed, to obtain various type definitions, enumerations, macros, etc. The files include `llnltyps.h, llnlmath.h, ida.h, nvector.h`, and one or more of the files `idadense.h, idaband.h, idaspgmr.h, idabbdpre.h` associated with the linear system solvers.
2. Set `N_Vector yy` and `N_Vector yp` to initial values for y and y', respectively, and set the problem size `Neq =` N. Depending on user options, also create the vector `id` of differential/algebraic component flags and/or the vector `constraints` of inequality constraint flags.
3. `idamem = IDAMalloc(Neq, resfn, rdata, t0, yy, yp, itol,...)` followed by `if (idamem == NULL) return(1)`, to allocate and initialize IDA's internal memory, and obtain a pointer to the IDA memory block. Here `rdata` is a pointer to a user-defined data block which is made available to the user residual function `resfn`. This call also includes tolerances `rtol, atol` and a tolerance type flag `itol`.
4. Specify the linear system solver to be used by making one of the calls:
   `flag = IDADense(...)` or `flag = IDABand(...)` or `flag = IDASpgmr(...)`
   followed by a test `if (flag != 0) return(1)`.
5. Optionally, correct the initial values in `yy, yp` with the call
   `flag = IDACalcIC(idamem, icopt, ...); if (flag != 0) return(1)`.
6. `flag = IDASolve(idamem, tout, tstop, &tt, yy, yp, itask)`
   (within a loop) to perform the integration of the DAE system from the current point to $t =$ `tout`. In this call, set `itask = NORMAL` to integrate past `tout` and interpolate. (Alternatively, set `itask = ONE_STEP` to take one step forward and return.) Following the call, process the vectors `yy` and `yp`, the computed solution y and y' at $t =$ `tt`. Also, test for the condition `flag < 0` to detect a failure.
7. `IDAFree(idamem)` to free the IDA memory block.
8. Supply a function `resfn` to define the DAE system residual function $F(t, y, y')$. This function is to have the form `resfn(Neq, tt, yy, yp, res, rdata)`. Likewise supply any routines for Jacobian evaluation or preconditioning, as required by the linear system solver chosen in step 4 above.

The call to `IDAMalloc` also includes integer and real arrays, `iopt` and `ropt`, devoted to optional inputs and outputs. These allow the user to supply certain optional inputs (e.g. the maximum method order or initial stepsize), and to obtain (as optional outputs) performance data (e.g. number of steps taken and stepsize last used). The details are given in a later subsection.

4.2. The User-Supplied Residual Function. The function $F(t, y, y')$ defining the DAE system is to be supplied by the user in the form of a C function, denoted `resfn` in the above Overview of Usage. The type and call list for this function must be as given by the following `typedef` (extracted from `ida.h`):

```
typedef int (*ResFn)(integer Neq, real tres, N_Vector yy,
              N_Vector yp, N_Vector resval, void *rdata);

/*****************************************************************
 *                                                               *
 * Type : ResFn                                                  *
 *---------------------------------------------------------------*
 * The F function which defines the DAE system   F(t,y,y')=0     *
 * must have type ResFn.                                         *
 * Symbols are as follows: t  <-> tres     y <-> yy              *
 *                         y' <-> yp       F <-> res (type ResFn) *
 * A ResFn takes as input the problem size Neq, the independent  *
 * variable value tres, the dependent variable vector yy, and the *
 * derivative (with respect to t) of the yy vector, yp.  It      *
 * stores the result of F(t,y,y') in the vector resval.  The     *
 * yy, yp, and resval arguments are of type N_Vector.            *
 * The rdata parameter is to be of the same type as the rdata    *
 * parameter passed by the user to the IDAMalloc routine. This   *
 * user-supplied pointer is passed to the user's res function    *
 * every time it is called, to provide access in res to user data.*
 *                                                               *
 * A ResFn res will return the value ires, which has possible    *
 * values RES_ERROR_RECVR = 1, RES_ERROR_NONRECVR = -1,          *
 * and SUCCESS = 0. The file ida.h may be used to obtain these   *
 * values but is not required; returning 0, +1, or -1 suffices.  *
 * RES_ERROR_NONRECVR will ensure that the program halts.        *
 * RES_ERROR_RECVR should be returned if, say, a yy or other input*
 * value is illegal. IDA will attempt to correct and retry.      *
 *                                                               *
 *****************************************************************/
```

4.3. Detailed Description of Callable Routines. In this subsection, we give complete user interface descriptions for the user-callable routines in the `IDA` module: `IDAMalloc`, `IDACalcIC`, `IDASolve`, and `IDAFree`. These are given in the form of exerpts from the header file `ida.h`. (Descriptions of the calls associated with the linear system solver are given in the next subsection.) Following these four descriptions is a list of the optional inputs and outputs associated with the `IDA` module.

In what follows, for each callable routine, the function declaration with arguments is followed by a summary of the routine, then a section of comments describing the call arguments, and finally a description of the possible return values. Note that in the code module itself, the variables t, y, and y' are generally denoted by `tt`, `yy`, and `yp`. The various constants that comprise allowed values for certain integer inputs, possible return values for the routines, and indices into `iopt` and `ropt` are defined in `ida.h`.

4.3.1. Memory allocation routine: IDAMalloc.

```
void *IDAMalloc(integer Neq, ResFn res, void *rdata, real t0,
      N_Vector y0, N_Vector yp0, int itol, real *rtol, void *atol,
      N_Vector id, N_Vector constraints, FILE *errfp, boole optIn,
      long int iopt[], real ropt[], void *machEnv);

 ****************************************************************/
 *                                                              *
 * Function : IDAMalloc                                         *
 *--------------------------------------------------------------*
 * IDAMalloc allocates and initializes memory for a problem to  *
 * to be solved by IDA.                                         *
 *                                                              *
 * Neq     is the number of equations in the DAE system.        *
 *         (In the parallel case, Neq is the global system size,*
 *         not the local size.)                                 *
 *                                                              *
 * res     is the residual function F in F(t,y,y') = 0.         *
 *                                                              *
 * rdata   is the data memory block (supplied by user) for res. *
 *         It is passed as a void pointer and is to be cast before*
 *         use in res.                                          *
 *                                                              *
 * t0      is the initial value of t, the independent variable. *
 *                                                              *
 * y0      is the initial condition vector y(t0).               *
 *                                                              *
 * yp0     is the initial condition vector y'(t0)               *
 *                                                              *
 * itol    is the type of tolerances to be used.                *
 *            The legal values are:                             *
 *               SS (scalar relative and absolute  tolerances), *
 *               SV (scalar relative tolerance and vector       *
 *                   absolute tolerance).                       *
 *                                                              *
 * rtol    is a pointer to the relative tolerance scalar.       *
 *                                                              *
 * atol    is a pointer (void) to the absolute tolerance scalar or*
 *            an N_Vector tolerance.                            *
 * (ewt)                                                        *
 *         Both rtol and atol are used to compute the error weight*
 *         vector, ewt. The error test required of a correction *
 *         delta is that the weighted-RMS norm of delta be less *
 *         than or equal to 1.0. Other convergence tests use the*
 *         same norm. The weighting vector used in this norm is *
 *         ewt. The components of ewt are defined by            *
 *         ewt[i] = 1.0/(rtol*yy[i] + atol[i]). Here, yy is the *
```

```
*         current approximate solution.  See the routine         *
*         N_VWrmsNorm for the norm used in this error test.      *
*                                                                *
* id      is an N_Vector, required conditionally, which states a *
*         given element to be either algebraic or differential.  *
*         A value of 1.0 indicates a differential variable while *
*         a 0.0 indicates an algebraic variable. 'id' is required*
*         if optional input SUPPRESSALG is set, or if IDACalcIC  *
*         is to be called with icopt = CALC_YA_YDP_INIT.         *
*         Otherwise, 'id' may be NULL.                           *
*                                                                *
* constraints  is an N_Vector defining inequality constraints    *
*         for each component of the solution vector y. If a given*
*         element of this vector has values +2 or -2, then the   *
*         corresponding component of y will be constrained to be *
*         > 0.0 or < 0.0, respectively, while if it is +1 or -1, *
*         the y component is constrained to be >= 0.0 or <= 0.0, *
*         respectively. If a component of constraints is 0.0,    *
*         then no constraint is imposed on the corresponding     *
*         component of y. The presence of a non-NULL constraints *
*         vector that is not 0.0 (ZERO) in all components will   *
*         cause constraint checking to be performed.             *
*                                                                *
* errfp   is the file pointer for an error file where all IDA    *
*            warning and error messages will be written. This    *
*            parameter can be stdout (standard output), stderr   *
*            (standard error), a file pointer (corresponding to  *
*            a user error file opened for writing) returned by   *
*            fopen, or NULL. If the user passes NULL, then all   *
*            messages will be written to standard output.        *
*                                                                *
* optIn   is a flag (boole) indicating whether there are any     *
*            optional inputs from the user in the arrays         *
*            iopt and iopt.                                      *
*            Pass FALSE to indicate no optional inputs and TRUE  *
*            to indicate that optional inputs are present.       *
*                                                                *
* iopt    is the user-allocated array (of size OPT_SIZE given    *
*            later) that will hold optional integer inputs and   *
*            outputs.  The user can pass NULL if he/she does not *
*            wish to use optional integer inputs or outputs.     *
*            If optIn is TRUE, the user should preset to 0 those *
*            locations for which default values are to be used.  *
*                                                                *
* ropt    is the user-allocated array (of size OPT_SIZE given    *
*            later) that will hold optional real inputs and      *
*            outputs.  The user can pass NULL if he/she does not *
*            wish to use optional real inputs or outputs.        *
```

```
*            If optIn is TRUE, the user should preset to 0.0 the *
*            locations for which default values are to be used.  *
*                                                                *
* machEnv is a pointer to a block of machine environment-specific*
*            information.  You may pass NULL for this argument   *
*            in a serial computer environment.                   *
*                                                                *
* Note: The tolerance values may be changed in between calls to  *
*       IDASolve for the same problem. These values refer to     *
*       (*rtol) and either (*atol), for a scalar absolute        *
*       tolerance, or the components of atol, for a vector       *
*       absolute tolerance.                                      *
*                                                                *
* If successful, IDAMalloc returns a pointer to initialized      *
* problem memory. This pointer should be passed to IDA. If       *
* an initialization error occurs, IDAMalloc prints an error      *
* message to the file specified by errfp and returns NULL.       *
*                                                                *
******************************************************************/
```

4.3.2. Initial condition calculation routine: IDACalcIC.

```
int IDACalcIC (void *ida_mem, int icopt, real tout1, real epicfac,
               int maxnh, int maxnj, int maxnit, int lsoff, real steptol);

******************************************************************/
*                                                                *
* Function : IDACalcIC                                           *
*                                                                *
*----------------------------------------------------------------*
* IDACalcIC calculates corrected initial conditions for the DAE  *
* system for a class of index-one problems of semi-implicit form.*
* It uses Newton iteration combined with a Linesearch algorithm. *
* Calling IDACalcIC is optional. It is only necessary when the   *
* initial conditions do not solve the given system.  I.e., if    *
* y0 and yp0 are known to satisfy F(t0, y0, yp0) = 0, then       *
* a call to IDACalcIC is NOT necessary (for index-one problems). *
*                                                                *
* A call to IDACalcIC must be preceded by a successful call to   *
* IDAMalloc, and by a successful call to the linear system       *
* solver specification routine.  In addition, IDACalcIC assumes  *
* that the vectors y0, yp0 and (if relevant) id and constraints  *
* that were passed to IDAMalloc remain unaltered since that call.*
*                                                                *
* The call to IDACalcIC should precede the call(s) to IDASolve   *
* for the given problem.                                         *
*                                                                *
```

```
* The arguments to IDACalcIC are as follows.  The first three -- *
* ida_mem, icopt, tout1 -- are required; the others are optional.*
* A zero value passed for any optional input specifies that the  *
* default value is to be used.                                   *
*                                                                *
* IDA_mem is the pointer to IDA memory returned by IDAMalloc.    *
*                                                                *
* icopt  is the option of IDACalcIC to be used.                  *
*        icopt = CALC_YA_YDP_INIT   directs IDACalcIC 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 was input to IDAMalloc, specifying  *
*                the differential and algebraic components.      *
*        icopt = CALC_Y_INIT   directs IDACalcIC to compute all  *
*                components of y, given y'.  id is not required. *
*                                                                *
* tout1  is the first value of t at which a soluton will be      *
*        requested (from IDASolve).  (This is needed here to     *
*        determine the direction of integration and rough scale  *
*        in the independent variable t.                          *
*                                                                *
* epicfac is a positive scalar factor in the Newton convergence  *
*        test.  This test uses a weighted RMS norm (with weights *
*        defined by the tolerances, as in IDASolve).  For new    *
*        initial value vectors y and y' to be accepted, the norm *
*        of J-inverse F(t0,y,y') is required to be less than     *
*        epicfac*0.33, where J is the system Jacobian.           *
*        The default is epicfac = .01, specified by passing 0.   *
*                                                                *
* maxnh  is the maximum number of values of h allowed in the     *
*        algorithm for icopt = CALC_YA_YDP_INIT, where h appears *
*        in the system Jacobian, J = dF/dy + (1/h)dF/dy'.        *
*        The default is maxnh = 5, specified by passing 0.       *
*                                                                *
* maxnj  is the maximum number of values of the approximate      *
*        Jacobian or preconditioner allowed, when the Newton     *
*        iterations appear to be slowly converging.              *
*        The default is maxnj = 4, specified by passing 0.       *
*                                                                *
* maxnit is the maximum number of Newton iterations allowed in   *
*        any one attempt to solve the IC problem.                *
*        The default is maxnit = 10, specified by passing 0.     *
*                                                                *
* lsoff  is an integer flag to turn off the linesearch algorithm *
*        (lsoff = 1). The default is lsoff = 0 (linesearch done).*
*                                                                *
* steptol is a positive lower bound on the norm of a Newton step.*
```

```
*        The default value is steptol = (unit roundoff)^(2/3),   *
*        specified by passing 0.                                 *
*                                                                *
*                                                                *
* IDACalcIC returns an int flag.  Its symbolic values and their  *
* meanings are as follows.  (The numerical return values are set *
* above in this file.)  All unsuccessful returns give a negative *
* return value.                                                  *
*                                                                *
* SUCCESS              IDACalcIC was successful.  The corrected  *
*                      initial value vectors are in y0 and yp0.  *
*                                                                *
* IC_IDA_NO_MEM        The argument ida_mem was NULL.            *
*                                                                *
* IC_ILL_INPUT         One of the input arguments was illegal.   *
*                      See printed message.                      *
*                                                                *
* IC_LINIT_FAIL        The linear solver's init routine failed.  *
*                                                                *
* IC_BAD_EWT           Some component of the error weight vector *
*                      is zero (illegal), either for the input   *
*                      value of y0 or a corrected value.         *
*                                                                *
* RES_NONRECOV_ERR     The user's ResFn residual routine returned*
*                      a non-recoverable error flag.             *
*                                                                *
* IC_FIRST_RES_FAIL    The user's ResFn residual routine returned*
*                      a recoverable error flag on the first call,*
*                      but IDACalcIC was unable to recover.      *
*                                                                *
* SETUP_FAILURE        The linear solver's setup routine had a   *
*                      non-recoverable error.                    *
*                                                                *
* SOLVE_FAILURE        The linear solver's solve routine had a   *
*                      non-recoverable error.                    *
*                                                                *
* IC_NO_RECOVERY       The user's residual routine, or the linear*
*                      solver's setup or solve routine had a     *
*                      recoverable error, but IDACalcIC was      *
*                      unable to recover.                        *
*                                                                *
* IC_FAILED_CONSTR     IDACalcIC was unable to find a solution   *
*                      satisfying the inequality constraints.    *
*                                                                *
* IC_FAILED_LINESRCH   The Linesearch algorithm failed to find a *
*                      solution with a step larger than steptol  *
*                      in weighted RMS norm.                     *
*                                                                *
```

```
 * IC_CONV_FAILURE     IDACalcIC failed to get convergence of the *
 *                     Newton iterations.                         *
 *                                                                *
 *                                                                *
 * The following optional outputs provided by IDACalcIC are       *
 * available in the iopt and ropt arrays.                         *
 *                                                                *
 * iopt[NRE]     = number of calls to the user residual function. *
 *                                                                *
 * iopt[NNI]     = number of Newton iterations performed.         *
 *                                                                *
 * iopt[NCFN]    = number of nonlinear convergence failures.      *
 *                                                                *
 * iopt[NSETUPS] = number of calls to linear solver setup routine.*
 *                                                                *
 * iopt[NBACKTR] = number of backtracks in Linesearch algorithm.  *
 *                                                                *
 * ropt[HUSED]   = value of h last used in the system Jacobian J  *
 *                 if icopt = CALC_YA_YDP_INIT.                   *
 *                                                                *
 ******************************************************************/
```

4.3.3. Main solver routine: IDASolve.

```
int IDASolve(void *ida_mem, real tout, real tstop, real *tret,
        N_Vector yret, N_Vector ypret, int itask);

 ******************************************************************/
 *                                                                *
 * Function : IDASolve                                            *
 *----------------------------------------------------------------*
 * IDASolve integrates the DAE over an interval in t, the         *
 * independent variable. If itask is NORMAL, then the solver      *
 * integrates from its current internal t value to a point at or  *
 * beyond tout, then interpolates to t = tout and returns y(tret) *
 * in the user-allocated vector yret. In general, tret = tout.    *
 * If itask is ONE_STEP, then the solver takes one internal step  *
 * of the independent variable and returns in yret the value of y *
 * at the new internal independent variable value. In this case,  *
 * tout is used only during the first call to IDASolve to         *
 * determine the direction of integration and the rough scale of  *
 * the problem. In either case, the independent variable value    *
 * reached by the solver is placed in (*tret). The user is        *
 * responsible for allocating the memory for this value. There are*
 * two other itask options: NORMAL_TSTOP and ONE_STEP_TSTOP. Each *
 * option acts as described above with the exception that the     *
 * solution does not go past the independent variable value tstop.*
```

```
*                                                               *
* IDA_mem is the pointer (void) to IDA memory returned by       *
* IDAMalloc.                                                    *
*                                                               *
* tout  is the next independent variable value  at which a      *
* computed solution is desired                                  *
*                                                               *
* tstop is the (optional) independent variable value past which *
* the solution is not to proceed (see itask options)            *
*                                                               *
* *tret  is the actual independent variable value corresponding *
* to the solution vector yret.                                  *
*                                                               *
* yret  is the computed solution vector. With no errors,        *
*           yret=y(tret).                                       *
*                                                               *
* ypret is the derivative of the computed solution vector at tret*
*                                                               *
* Note: yret and ypret may be the same N_Vectors as y0 and yp0  *
* in the call to IDAMalloc.                                     *
*                                                               *
* itask is NORMAL, NORMAL_TSTOP, ONE_STEP, or ONE_STEP_TSTOP.   *
*        These modes are described above.                       *
*                                                               *
*****************************************************************/
```

The following is a list of the possible return values for the routine `IDASolve`. The first three are for successful cases, and the rest are for a variety of possible failures.

```
*****************************************************************/
*                                                               *
* The return values for IDASolve are described below.           *
* (The numerical return values are defined above in this file.) *
* All unsuccessful returns give a negative return value.        *
*                                                               *
* NORMAL_RETURN      : IDASolve succeeded.                      *
*                                                               *
* INTERMEDIATE_RETURN: IDASolve returns computed results for the *
*                  last single step (itask = ONE_STEP).         *
*                                                               *
* TSTOP_RETURN       : IDASolve returns computed results for the *
*                  independent variable value tstop. That is,   *
*                  tstop was reached.                           *
*                                                               *
* IDA_NO_MEM         : The IDA_mem argument was NULL.           *
*                                                               *
* ILL_INPUT          : One of the inputs to IDASolve is illegal. *
```

```
*                 This includes the situation when a component   *
*                 of the error weight vectors becomes < 0 during *
*                 internal stepping. The ILL_INPUT flag          *
*                 will also be returned if the linear solver     *
*                 routine IDA--- (called by the user after       *
*                 calling IDAMalloc) failed to set one of the    *
*                 linear solver-related fields in IDA_mem or     *
*                 if the linear solver's init routine failed. In *
*                 any case, the user should see the printed      *
*                 error message for more details.                *
*                                                                *
* TOO_MUCH_WORK      : The solver took mxstep internal steps but *
*                 could not reach tout. The default value for    *
*                 mxstep is MXSTEP_DEFAULT = 500.                *
*                                                                *
* TOO_MUCH_ACC       : The solver could not satisfy the accuracy *
*                 demanded by the user for some internal step.   *
*                                                                *
* ERR_FAILURE        : Error test failures occurred too many     *
*                 times (=MXETF = 10) during one internal step.  *
*                                                                *
* CONV_FAILURE       : Convergence test failures occurred too    *
*                 many times (= MXNCF = 10) during one internal  *
*                 step.                                          *
*                                                                *
* SETUP_FAILURE      : The linear solver's setup routine failed  *
*                 in an unrecoverable manner.                    *
*                                                                *
* SOLVE_FAILURE      : The linear solver's solve routine failed  *
*                 in an unrecoverable manner.                    *
*                                                                *
* CONSTR_FAILURE     : The inequality constraints were violated, *
*                  and the solver was unable to recover.         *
*                                                                *
* REP_RES_REC_ERR    : The user's residual function repeatedly   *
*                  returned a recoverable error flag, but the    *
*                  solver was unable to recover.                 *
*                                                                *
* RES_NONRECOV_ERR   : The user's residual function returned a   *
*                  nonrecoverable error flag.                    *
*                                                                *
******************************************************************/
```

4.3.4. Deallocation routine: IDAFree.

```
void IDAFree(void *ida_mem);

/*****************************************************************
 *                                                               *
 * Function : IDAFree                                            *
 *---------------------------------------------------------------*
 * IDAFree frees the problem memory IDA_mem allocated by         *
 * IDAMalloc.  Its only argument is the pointer idamem           *
 * returned by IDAMalloc.                                        *
 *                                                               *
 *****************************************************************/
```

4.3.5. Optional input and output arrays: `iopt, ropt.` The communication of several optional input parameters to `IDAMalloc` is handled by placing their values in appropriate elements of either of the arrays `iopt` or `ropt`. Also, numerous optional outputs are available in these two arrays, following a call to either `IDACalcIC` or `IDASolve`. These optional inputs and optional outputs and their meanings are given below. In the case of optional inputs, a value of 0 causes IDA to use the default value for that input, and if any non-default values are to be used, the input flag `optIn` to `IDAMalloc` must be TRUE.

These two arrays are also used for optional outputs from the linear solver modules, as described in the next subsection.

```
 *****************************************************************/
 *                                                               *
 * Optional Inputs and Outputs                                   *
 *---------------------------------------------------------------*
 * The user should declare two arrays for optional inputs to     *
 * IDAMalloc and optional outputs from IDACalcIC and IDASolve:   *
 * a long int array iopt for optional integer input/output       *
 * and a real array ropt for optional real input/output.         *
 * The size of both these arrays should be OPT_SIZE.             *
 * So the user's declaration should look like:                   *
 *   long int iopt[OPT_SIZE];                                    *
 *   real     ropt[OPT_SIZE];                                    *
 *                                                               *
 * The enumerations listed earlier are indices into the          *
 * iopt and ropt arrays. Here is a brief description of the      *
 * contents of these positions:                                  *
 *                                                               *
 * iopt[MAXORD] : maximum order to be used by the solver.        *
 *                Optional input. (Default = 5)                  *
 *                                                               *
 * iopt[MXSTEP] : maximum number of internal steps to be taken by *
 *                the solver in its attempt to reach tout.       *
```

```
*                Optional input. (Default = 500).                *
*                                                                *
* iopt[SUPPRESSALG]: flag to indicate whether or not to suppress *
*                algebraic variables in the local error tests:   *
*                0 = do not suppress ; 1 = do suppress;          *
*                the default is 0.  Optional input.              *
*                NOTE: if suppressed algebraic variables is      *
*                selected, the nvector 'id' must be supplied for *
*                identification of those algebraic components.   *
*                                                                *
* iopt[NST]    : cumulative number of internal steps taken by    *
*                the solver (total so far).  Optional output.    *
*                                                                *
* iopt[NRE]    : number of calls to the user's residual function.*
*                Optional output.                                *
*                                                                *
* iopt[NNI]     : number of Newton iterations performed.         *
*                 Optional output.                               *
*                                                                *
* iopt[NCFN]    : number of nonlinear convergence failures       *
*                 that have occurred. Optional output.           *
*                                                                *
* iopt[NETF]    : number of local error test failures that       *
*                 have occurred. Optional output.                *
*                                                                *
* iopt[NSETUPS] : number of calls to lsetup routine.             *
*                                                                *
* iopt[NBACKTR] : number of backtrack operations done in the     *
*                 linesearch algorithm in IDACalcIC.             *
*                                                                *
* iopt[KUSED]   : order used during the last internal step.      *
*                 Optional output.                               *
*                                                                *
* iopt[KNEXT]   : order to be used on the next internal step.    *
*                 Optional output.                               *
*                                                                *
* iopt[LENRW]   : size of required IDA internal real work        *
*                 space, in real words.  Optional output.        *
*                                                                *
* iopt[LENIW]   : size of required IDA internal integer work     *
*                 space, in integer words.  Optional output.     *
*                                                                *
* ropt[HINIT]   : initial step size. Optional input.             *
*                                                                *
* ropt[HMAX]    : maximum absolute value of step size allowed.   *
*                 Optional input. (Default is infinity).         *
*                                                                *
* ropt[NCONFAC] : factor in nonlinear convergence test for use   *
```

```
*                 during integration.  Optional input.          *
*                 The default value is 1.                       *
*                                                               *
* ropt[HUSED]   : step size for the last internal integration   *
*                 step (if from IDASolve), or the last value of *
*                 the artificial step size h (if from IDACalcIC).*
*                 Optional output.                              *
*                                                               *
* ropt[HNEXT]   : step size to be attempted on the next internal *
*                 step. Optional output.                        *
*                                                               *
* ropt[TNOW]    : current internal independent variable value   *
*                 reached by the solver.  Optional output.      *
*                                                               *
* ropt[TOLSF]   : a suggested factor by which the user's        *
*                 tolerances should be scaled when too much     *
*                 accuracy has been requested for some internal *
*                 step. Optional output.                        *
*                                                               *
****************************************************************/
```

4.4. Specifying the Linear Solver. At present, there are three linear solver modules from which the IDA user can select: IDADENSE, IDABAND, and IDASPGMR. These contain the dense and band direct solvers, and the SPGMR method (Krylov iterative) solver. The selection is made in the user program by making a call to the appropriate routine: `IDADense`, `IDABand`, or `IDASpgmr`. In the direct cases, the user may (optionally) supply a routine that computes the system Jacobian; otherwise this is done by an internal difference-quotient routine. In the SPGMR case, there are two optional user-supplied routines related to preconditioning: one for evaluation of the preconditioner matrix, and one for the solution of the associated linear systems.

There are two important points to remember when supplying a Jacobian or preconditioner for use with an IDA linear solver module. First, it is generally sufficient to provide only a crude approximation to the true system Jacobian $J = \partial F/\partial y + \alpha \partial F/\partial y'$. This is because the user-supplied matrix is used within a Newton iteration (and in the Krylov case, also within a Krylov iteration). Hence a less expensive approximate Jacobian, resulting in a small increase in the number of these outer iterations, is usually a good tradeoff. On the other hand, the user routine which computes the approximate Jacobian or preconditioner (if supplied) is not called at every Newton iteration, nor even on every time step, but less frequently. Thus it is appropriate to go to some expense in this routine if it reduces the cost of the subsequent linear system solve operations significantly. Some experimentation may be needed to find the best compromise between a Jacobian or preconditioner that is expensive (but accurate) and one that is inexpensive (but results in slow convergence).

For each linear solver module, we give below a full description of the user-callable routine (function declaration followed by a description of the arguments), the associated optional user-supplied routines for Jacobian evaluation or preconditioning, and the optional outputs

associated with the linear solver module. These consist of excerpts from the header file for the module. In the descriptions below, the scalar α is denoted `cj`.

For each linear solver module, the user can pass a pointer to a user-defined block of data, denoted `jdata` or `pdata` below, associated with the evaluation of the Jacobian (direct cases) or the treatment of the preconditioner (Krylov case). This pointer is passed back to the user-supplied routine(s), if any, associated with that linear solver module, in order to access problem data needed there.

4.4.1. The dense solver.

```
int IDADense(void *IDA_mem, IDADenseJacFn djac, void *jdata);

/*****************************************************************
 *                                                               *
 * Function : IDADense                                           *
 *---------------------------------------------------------------*
 * A call to the IDADense function links the main IDA integrator *
 * with the IDADENSE linear solver module.                       *
 *                                                               *
 * IDA_mem is the pointer to IDA memory returned by IDAMalloc.   *
 *                                                               *
 * djac is the dense Jacobian approximation routine to be used.  *
 *         A user-supplied djac routine must be of type          *
 *         IDADenseJacFn (see below).  Pass NULL for djac if IDA *
 *         is to use the default difference quotient routine     *
 *         IDADenseDQJac supplied with this module.              *
 *                                                               *
 * jdata is a pointer to user data which is passed to the djac   *
 *         routine every time it is called.                      *
 *                                                               *
 * IDADense returns either                                       *
 *     SUCCESS = 0         if successful, or                     *
 *     IDA_DENSE_FAIL = -1 if either IDA_mem was null, or a      *
 *                         malloc failure occurred.              *
 *****************************************************************/
```

If the user chooses to provide a dense approximate Jacobian to IDA, it must be supplied in the form of a C routine conforming to the following `typedef`.

```
typedef int (*IDADenseJacFn)(integer Neq, real tt, N_Vector yy, N_Vector yp,
                  real cj, N_Vector constraints, ResFn res, void *rdata,
                  void *jdata, N_Vector resvec, N_Vector ewt, real hh,
                  real uround, DenseMat JJ, long int *nrePtr,
                  N_Vector tempv1, N_Vector tempv2, N_Vector tempv3);

/*****************************************************************
 *                                                               *
```

```
* Type : IDADenseJacFn                                               *
*-------------------------------------------------------------------*
* A dense Jacobian approximation function djac must have the        *
* prototype given below. Its parameters are:                        *
*                                                                   *
* Neq is the problem size, and length of all vector arguments.      *
*                                                                   *
* tt  is the current value of the independent variable t.           *
*                                                                   *
* yy  is the current value of the dependent variable vector,        *
*       namely the predicted value of y(t).                         *
*                                                                   *
* yp  is the current value of the derivative vector y',             *
*       namely the predicted value of y'(t).                        *
*                                                                   *
* cj  is the scalar in the system Jacobian, proportional to 1/hh.   *
*                                                                   *
* constraints is the vector of inequality constraint options        *
*             (as passed to IDAMalloc).  Included here to allow     *
*             for checking of incremented y values in difference    *
*             quotient calculations.                                *
*                                                                   *
* res    is the residual function for the DAE problem.              *
*                                                                   *
* rdata  is a pointer to user data to be passed to res, the same    *
*       as the rdata parameter passed to IDAMalloc.                 *
*                                                                   *
* jdata  is a pointer to user Jacobian data - the same as the       *
*       jdata parameter passed to IDADense.                         *
*                                                                   *
* resvec is the residual vector F(tt,yy,yp).                        *
*                                                                   *
* ewt    is the error weight vector.                                *
*                                                                   *
* hh     is a tentative step size in t.                             *
*                                                                   *
* uround is the machine unit roundoff.                              *
*                                                                   *
* JJ     is the dense matrix (of type DenseMat) to be loaded by     *
*       an IDADenseJacFn routine with an approximation to the       *
*       system Jacobian matrix                                      *
*             J = dF/dy + cj*dF/dy'                                 *
*       at the given point (t,y,y'), where the DAE system is        *
*       given by F(t,y,y') = 0.  JJ is preset to zero, so only      *
*       the nonzero elements need to be loaded.  See note below.    *
*                                                                   *
* nrePtr is a pointer to the memory location containing the         *
*       IDA problem data nre = number of calls to res. This         *
```

```
 *        Jacobian routine should update this counter by adding   *
 *        on the number of res calls it makes in order to         *
 *        approximate the Jacobian, if any.  For example, if this *
 *        routine calls res a total of Neq times, then it should  *
 *        perform the update *nrePtr += Neq.                      *
 *                                                                *
 * tempv1, tempv2, tempv3 are pointers to memory allocated for    *
 *        N_Vectors which can be used by an IDADenseJacFn routine *
 *        as temporary storage or work space.                     *
 *                                                                *
 * Note: The following are two efficient ways to load JJ:         *
 * (1) (with macros - no explicit data structure references)      *
 *     for (j=0; j < Neq; j++) {                                  *
 *       col_j = DENSE_COL(JJ,j);                                 *
 *       for (i=0; i < Neq; i++) {                                *
 *         generate J_ij = the (i,j)th Jacobian element           *
 *         col_j[i] = J_ij;                                       *
 *       }                                                        *
 *     }                                                          *
 * (2) (without macros - explicit data structure references)      *
 *     for (j=0; j < Neq; j++) {                                  *
 *       col_j = (JJ->data)[j];                                   *
 *       for (i=0; i < Neq; i++) {                                *
 *         generate J_ij = the (i,j)th Jacobian element           *
 *         col_j[i] = J_ij;                                       *
 *       }                                                        *
 *     }                                                          *
 * A third way, using the DENSE_ELEM(A,i,j) macro, is much less   *
 * efficient in general.  It is only appropriate for use in small *
 * problems in which efficiency of access is NOT a major concern. *
 *                                                                *
 * The IDADenseJacFn should return                                *
 *     0 if successful,                                           *
 *     a positive int if a recoverable error occurred, or         *
 *     a negative int if a nonrecoverable error occurred.         *
 * In the case of a recoverable error return, IDA will attempt to *
 * recover by reducing the stepsize (which changes cj).           *
 ******************************************************************/
```

Optional outputs associated with IDADENSE are available in the `iopt` array, as follows:

```
/*****************************************************************
 *                                                               *
 * IDADENSE solver optional output indices                       *
 *---------------------------------------------------------------*
 * The following enumeration gives a symbolic name to each       *
 * IDADENSE optional output. The symbolic names are used as      *
```

```
 * indices into the iopt and ropt arrays passed to IDAMalloc.     *
 * The IDADENSE optional outputs are:                              *
 *                                                                 *
 * iopt[DENSE_NJE] : number of Jacobian evaluations, i.e. of       *
 *                   calls made to the dense Jacobian routine      *
 *                   (default or user-supplied).                   *
 *                                                                 *
 * iopt[DENSE_LRW] : size (in real words) of real workspace        *
 *                   matrices and vectors used by this module.     *
 *                                                                 *
 * iopt[DENSE_LIW] : size (in integer words) of integer            *
 *                   workspace vectors used by this module.        *
 *                                                                 *
 *****************************************************************/
```

4.4.2. The band solver.

```
int IDABand(void *IDA_mem, integer mupper, integer mlower,
            IDABandJacFn bjac, void *jdata);

/*****************************************************************
 *                                                               *
 * Function : IDABand                                            *
 *---------------------------------------------------------------*
 * A call to the IDABand function links the main IDA integrator  *
 * with the IDABAND linear solver module.                        *
 *                                                               *
 * IDA_mem is the pointer to IDA memory returned by IDAMalloc.   *
 *                                                               *
 * mupper is the upper bandwidth of the banded Jacobian matrix.  *
 *                                                               *
 * mlower is the lower bandwidth of the banded Jacobian matrix.  *
 *                                                               *
 * bjac is the banded Jacobian approximation routine to be used. *
 *         A user-supplied bjac routine must be of type          *
 *         IDABandJacFn (see below).  Pass NULL for bjac if IDA  *
 *         is to use the default difference quotient routine     *
 *         IDABandDQJac supplied with this module.               *
 *                                                               *
 * jdata is a pointer to user data which is passed to the bjac   *
 *         routine every time it is called.                      *
 *                                                               *
 * IDABand returns either                                        *
 *    SUCCESS = 0            if successful, or                   *
 *    IDA_BAND_FAIL = -1     if either IDA_mem was NULL or a     *
 *                           malloc failure occurred, or         *
 *    IDA_BAND_BAD_ARG = -2  if mupper or mlower is illegal.     *
```

```
 ****************************************************************/
```

In the call to `IDABand`, the half-bandwidths `mupper` and `mlower` need not be the true half-bandwidths of the system Jacobian for the DAE problem. Smaller values may greatly reduce the expense of Jacobian evaluation and band matrix computations.

If the user chooses to provide a banded approximate Jacobian to IDA, it must be supplied in the form of a C routine conforming to the following `typedef`.

```
typedef int (*IDABandJacFn)(integer Neq, integer mupper, integer mlower,
                  real tt, N_Vector yy, N_Vector yp, real cj,
                  N_Vector constraints, ResFn res, void *rdata, void *jdata,
                  N_Vector resvec, N_Vector ewt, real hh, real uround,
                  BandMat JJ, long int *nrePtr, N_Vector tempv1,
                  N_Vector tempv2, N_Vector tempv3);

/****************************************************************
 *                                                              *
 * Type : IDABandJacFn                                          *
 *--------------------------------------------------------------*
 * A banded Jacobian approximation function bjac must have the  *
 * prototype given below. Its parameters are:                   *
 *                                                              *
 * Neq is the problem size, and length of all vector arguments. *
 *                                                              *
 * mupper is the upper bandwidth of the banded Jacobian matrix. *
 *                                                              *
 * mlower is the lower bandwidth of the banded Jacobian matrix. *
 *                                                              *
 * tt  is the current value of the independent variable t.      *
 *                                                              *
 * yy  is the current value of the dependent variable vector,   *
 *       namely the predicted value of y(t).                    *
 *                                                              *
 * yp  is the current value of the derivative vector y',        *
 *       namely the predicted value of y'(t).                   *
 *                                                              *
 * cj  is the scalar in the system Jacobian, proportional to 1/hh.*
 *                                                              *
 * constraints is the vector of inequality constraint options   *
 *             (as passed to IDAMalloc).  Included here to allow *
 *             for checking of incremented y values in difference *
 *             quotient calculations.                           *
 *                                                              *
 * res    is the residual function for the DAE problem.         *
 *                                                              *
 * rdata  is a pointer to user data to be passed to res, the same *
 *       as the rdata parameter passed to IDAMalloc.            *
```

```
*                                                               *
* jdata  is a pointer to user Jacobian data - the same as the   *
*        jdata parameter passed to IDABand.                     *
*                                                               *
* resvec is the residual vector F(tt,yy,yp).                    *
*                                                               *
* ewt    is the error weight vector.                            *
*                                                               *
* hh     is a tentative step size in t.                         *
*                                                               *
* uround is the machine unit roundoff.                          *
*                                                               *
* JJ     is the band matrix (of type BandMat) to be loaded by   *
*        an IDABandJacFn routine with an approximation to the   *
*        system Jacobian matrix                                 *
*              J = dF/dy + cj*dF/dy'                            *
*        at the given point (t,y,y'), where the DAE system is   *
*        given by F(t,y,y') = 0.  JJ is preset to zero, so only *
*        the nonzero elements need to be loaded.  See note below.*
*                                                               *
* nrePtr is a pointer to the memory location containing the     *
*        IDA problem data nre = number of calls to res. This    *
*        Jacobian routine should update this counter by adding  *
*        on the number of res calls it makes in order to        *
*        approximate the Jacobian, if any.  For example, if this *
*        routine calls res a total of M times, then it should   *
*        perform the update *nrePtr += M.                       *
*                                                               *
* tempv1, tempv2, tempv3 are pointers to memory allocated for   *
*        N_Vectors which can be used by an IDABandJacFn routine *
*        as temporary storage or work space.                    *
*                                                               *
*                                                               *
* Note: The following are two efficient ways to load JJ:        *
*                                                               *
* (1) (with macros - no explicit data structure references)     *
*    for (j=0; j < Neq; j++) {                                  *
*       col_j = BAND_COL(JJ,j);                                 *
*       for (i=j-mupper; i <= j+mlower; i++) {                  *
*         generate J_ij = the (i,j)th Jacobian element          *
*         BAND_COL_ELEM(col_j,i,j) = J_ij;                      *
*       }                                                       *
*     }                                                         *
*                                                               *
* (2) (with BAND_COL macro, but without BAND_COL_ELEM macro)    *
*    for (j=0; j < Neq; j++) {                                  *
*       col_j = BAND_COL(JJ,j);                                 *
*       for (k=-mupper; k <= mlower; k++) {                     *
```

```
 *         generate J_ij = the (i,j)th Jacobian element, i=j+k    *
 *         col_j[k] = J_ij;                                       *
 *       }                                                        *
 *     }                                                          *
 *                                                                *
 * A third way, using the BAND_ELEM(A,i,j) macro, is much less    *
 * efficient in general.  It is only appropriate for use in small *
 * problems in which efficiency of access is NOT a major concern. *
 *                                                                *
 * The IDABandJacFn should return                                 *
 *     0 if successful,                                           *
 *     a positive int if a recoverable error occurred, or         *
 *     a negative int if a nonrecoverable error occurred.         *
 * In the case of a recoverable error return, IDA will attempt to *
 * recover by reducing the stepsize (which changes cj).           *
 ******************************************************************/
```

Optional outputs associated with IDABAND are available in the `iopt` array, as follows:

```
/*****************************************************************
 *                                                               *
 * IDABAND solver optional output indices                        *
 *---------------------------------------------------------------*
 * The following enumeration gives a symbolic name to each       *
 * IDABAND optional output. The symbolic names are used as       *
 * indices into the iopt and ropt arrays passed to IDAMalloc.    *
 * The IDABAND optional outputs are:                             *
 *                                                               *
 * iopt[BAND_NJE] : number of Jacobian evaluations, i.e. of      *
 *                  calls made to the band Jacobian routine      *
 *                  (default or user-supplied).                  *
 *                                                               *
 * iopt[BAND_LRW] : size (in real words) of real workspace       *
 *                  matrices and vectors used by this module.    *
 *                                                               *
 * iopt[BAND_LIW] : size (in integer words) of integer           *
 *                  workspace vectors used by this module.       *
 *                                                               *
 *****************************************************************/
```

4.4.3. The SPGMR solver.

```
int IDASpgmr(void *IDA_mem, IDASpgmrPrecondFn precond,
             IDASpgmrPSolveFn psolve, int gstype, int maxl, int maxrs,
             real eplifac, real dqincfac, void *pdata);

/*****************************************************************
```

```
*                                                               *
* Function : IDASpgmr                                           *
*---------------------------------------------------------------*
* A call to the IDASpgmr function links the main IDA integrator *
* with the IDASPGMR linear solver module.  Its parameters are   *
* as follows:                                                   *
*                                                               *
* IDA_mem   is the pointer to IDA memory returned by IDAMalloc. *
*                                                               *
* precond   is the user's preconditioner setup routine. It is   *
*           used to evaluate and preprocess any Jacobian-related*
*           data needed by the psolve routine.  See the         *
*           description of the type IDASpgmrPrecondFn above.    *
*           Pass NULL if no such data setup is required.        *
*                                                               *
* psolve    is the user's preconditioner solve routine. It is   *
*           used to solve linear systems P z = r, where P is the*
*           preconditioner matrix.  See the description of the  *
*           type IDASpgmrPSolveFn above.  Pass NULL for psolve  *
*           if no preconditioning is to be done.  However, a    *
*           preconditioner of some form is strongly encouraged. *
*                                                               *
* gstype    is the type of Gram-Schmidt orthogonalization to be *
*           used.  This must be one of the two enumeration      *
*           constants MODIFIED_GS or CLASSICAL_GS defined in    *
*           iterativ.h.  These correspond to using modified or  *
*           classical Gram-Schmidt algorithms, respectively.    *
*                                                               *
* maxl      is the maximum Krylov subspace dimension, an        *
*           optional input.  Pass 0 to use the default value,   *
*           MIN(Neq, 5).  Otherwise pass a positive integer.    *
*                                                               *
* maxrs     is the maximum number of restarts to be used in the *
*           GMRES algorithm, an optional input.  maxrs must be a*
*           non-negative integer, or -1.  Pass 0 to use the     *
*           default value, which is 5.  Pass -1 to use the      *
*           value 0, meaning no restarts.  In any case, maxrs   *
*           will be restricted to the range 0 to Neq/maxl.      *
*                                                               *
* eplifac   is a factor in the linear iteration convergence     *
*           test constant, an optional input.  Pass 0.0 to use  *
*           the default, which is 1.0.  Otherwise eplifac must  *
*           be a positive real number.                          *
*                                                               *
* dqincfac  is a factor in the increments to yy used in the     *
*           difference quotient approximations to matrix-vector *
*           products Jv, an optional input.  Pass 0.0 to use    *
*           the default, which is 1.0.  Otherwise dqincfac must *
```

```
 *           be a positive real number.                          *
 *                                                               *
 * pdata     is a pointer to user preconditioner data.  This     *
 *           pointer is passed to precond and psolve every time  *
 *           these routines are called.                          *
 *                                                               *
 * IDASpgmr returns either                                       *
 *    SUCCESS = 0             if successful, or                  *
 *    IDA_SPGMR_FAIL = -1     if either IDA_mem was null or a    *
 *                            malloc failure occurred, or        *
 *    IDA_SPGMR_BAD_ARG = -2  if gstype was found illegal.       *
 ***************************************************************/
```

Preconditioning is an important part of using IDA with the SPGMR solver (or any Krylov solver). In any nontrivial DAE problem, it is usually essential to provide a preconditioner of some sort. This is primarily because the Krylov iteration convergence test is made on the preconditioned residual vector. Without preconditioning, i.e. if the preconditioner is the identity matrix, this test can be a very poor measure of convergence, and may not even be dimensionally consistent, meaning that the components with different physical units are being compared as dimensionless numbers.

In supplying a preconditioner for IDASPGMR, the user must supply a C routine (denoted `PSolve` below) of type `IDASpgmrPSolveFn`, as given in the following `typedef`. Typically, the preconditioner is based on some (possibly crude) approximation to the system Jacobian J. Usually, once the preconditioner matrix P is calculated, and possibly preprocessed, it is beneficial to save the resulting matrix for use over many iterations (over several time steps). To do that, the user must also supply a C routine (denoted `Precond`) of type `IDASpgmrPrecondFn`, as given in the next `typedef` following. The `Precond` routine is called relatively infrequently, while the `PSolve` routine is called at every Krylov iteration.

```
typedef int (*IDASpgmrPSolveFn)(integer Neq, real tt, N_Vector yy,
             N_Vector yp, N_Vector rr, real cj, ResFn res, void *rdata,
             void *pdata, N_Vector ewt, real delta, N_Vector rvec,
             N_Vector zvec, long int *nrePtr, N_Vector tempv);

/***************************************************************
 *                                                             *
 * Type : IDASpgmrPSolveFn                                     *
 *-------------------------------------------------------------*
 * The optional user-supplied function PSolve must compute a   *
 * solution to the linear system P z = r, where P is the left  *
 * preconditioner defined by the user.  If no preconditioning  *
 * is desired, pass NULL for PSolve to IDASpgmr.               *
 *                                                             *
 * A preconditioner solve function PSolve must have the        *
 * prototype given below.  Its parameters are as follows:      *
 *                                                             *
```

```
* Neq is the problem size, and length of all vector arguments.   *
*                                                                *
* tt  is the current value of the independent variable t.        *
*                                                                *
* yy  is the current value of the dependent variable vector y.   *
*                                                                *
* yp  is the current value of the derivative vector y'.          *
*                                                                *
* rr  is the current value of the residual vector F(t,y,y').     *
*                                                                *
* cj  is the scalar in the system Jacobian, proportional to 1/hh.*
*                                                                *
* res    is the residual function for the DAE problem.           *
*                                                                *
* rdata  is a pointer to user data to be passed to res, the same *
*        as the rdata parameter passed to IDAMalloc.             *
*                                                                *
* pdata  is a pointer to user preconditioner data - the same as  *
*        the pdata parameter passed to IDASpgmr.                 *
*                                                                *
* ewt    is the input error weight vector (see delta below).     *
*                                                                *
* delta  is an input tolerance for use by PSolve if it uses an   *
*        iterative method in its solution.   In that case, the   *
*        the residual vector r - P z of the system should be     *
*        made less than delta in weighted L2 norm, i.e.,         *
*            sqrt [ Sum (Res[i]*ewt[i])^2 ] < delta .            *
*                                                                *
* rvec   is the input right-hand side vector r.                  *
*                                                                *
* zvec   is the computed solution vector z.                      *
*                                                                *
* nrePtr is a pointer to the memory location containing the      *
* IDA problem data nre = number of calls to res.  This PSolve    *
* routine should update the counter nre by adding on the number  *
* of res calls it makes in order to compute z, if any.           *
* Thus if this routine calls res a total of W times, it should   *
* perform the update *nrePtr += W.                               *
*                                                                *
* tempv  is an N_Vector which can be used by the PSolve          *
* routine as temporary storage or work space.                    *
*                                                                *
*                                                                *
* The IDASpgmrPSolveFn should return                             *
*     0 if successful,                                           *
*     a positive int if a recoverable error occurred, or         *
*     a negative int if a nonrecoverable error occurred.         *
* Following a recoverable error, IDA will attempt to recover by  *
```

```
 * updating the preconditioner and/or reducing the stepsize.      *
 ****************************************************************/


typedef int (*IDASpgmrPrecondFn)(integer Neq, real tt, N_Vector yy,
        N_Vector yp, N_Vector rr, real cj, ResFn res,
        void *rdata, void *pdata, N_Vector ewt, N_Vector constraints,
        real hh, real uround,  long int *nrePtr, N_Vector tempv1,
        N_Vector tempv2, N_Vector tempv3);

/****************************************************************
 *                                                              *
 * Type : IDASpgmrPrecondFn                                     *
 *--------------------------------------------------------------*
 * The optional user-supplied functions Precond and PSolve      *
 * together must define the left preconditoner matrix P         *
 * approximating the system Jacobian matrix                     *
 *    J = dF/dy + cj*dF/dy'                                     *
 * (where the DAE system is F(t,y,y') = 0), and solve the linear *
 * systems P z = r.   Precond is to do any necessary setup      *
 * operations, and PSolve is to compute the solution of P z = r. *
 *                                                              *
 * The preconditioner setup function Precond is to evaluate and *
 * preprocess any Jacobian-related data needed by the           *
 * preconditioner solve function PSolve.  This might include    *
 * forming a crude approximate Jacobian, and performing an LU   *
 * factorization on it.  This function will not be called in    *
 * advance of every call to PSolve, but instead will be called  *
 * only as often as necessary to achieve convergence within the *
 * Newton iteration in IDA.  If the PSolve function needs no    *
 * preparation, the Precond function can be NULL.               *
 *                                                              *
 * Each call to the Precond function is preceded by a call to   *
 * the system function res with the same (t,y,y') arguments.    *
 * Thus the Precond function can use any auxiliary data that is *
 * computed and saved by the res function and made accessible   *
 * to Precond.                                                  *
 *                                                              *
 * The error weight vector ewt, step size hh, and unit roundoff *
 * uround are provided to the Precond function for possible use *
 * in approximating Jacobian data, e.g. by difference quotients. *
 *                                                              *
 * A preconditioner setup function Precond must have the        *
 * prototype given below.  Its parameters are as follows:       *
 *                                                              *
 * Neq is the problem size, and length of all vector arguments. *
 *                                                              *
 * tt  is the current value of the independent variable t.      *
```

```
*                                                               *
* yy  is the current value of the dependent variable vector,    *
*       namely the predicted value of y(t).                     *
*                                                               *
* yp  is the current value of the derivative vector y',         *
*       namely the predicted value of y'(t).                    *
*                                                               *
* rr  is the current value of the residual vector F(t,y,y').    *
*                                                               *
* cj  is the scalar in the system Jacobian, proportional to 1/hh.*
*                                                               *
* res    is the residual function for the DAE problem.          *
*                                                               *
* rdata  is a pointer to user data to be passed to res, the same *
*       as the rdata parameter passed to IDAMalloc.             *
*                                                               *
* pdata  is a pointer to user preconditioner data - the same as  *
*       the pdata parameter passed to IDASpgmr.                 *
*                                                               *
* ewt    is the error weight vector.                            *
*                                                               *
* constraints  is the constraints vector.                       *
*                                                               *
* hh     is a tentative step size in t.                         *
*                                                               *
* uround is the machine unit roundoff.                          *
*                                                               *
* nrePtr is a pointer to the memory location containing the     *
* IDA problem data nre = number of calls to res.  This Precond  *
* routine should update the counter nre by adding on the number  *
* of res calls it makes in order to compute P, if any.          *
* Thus if this routine calls res a total of W times, it should  *
* perform the update *nrePtr += W.                              *
*                                                               *
* tempv1, tempv2, tempv3 are pointers to vectors of type        *
* N_Vector which can be used by an IDASpgmrPrecondFn routine as  *
* temporary storage or work space.                              *
*                                                               *
*                                                               *
* The IDASpgmrPrecondFn should return                           *
*     0 if successful,                                          *
*     a positive int if a recoverable error occurred, or        *
*     a negative int if a nonrecoverable error occurred.        *
* In the case of a recoverable error return, IDA will attempt to *
* recover by reducing the stepsize (which changes cj).          *
****************************************************************/
```

Optional outputs associated with IDASPGMR are available in the `iopt` array, as follows:

```
/*************************************************************
 *                                                           *
 * IDASPGMR solver optional output indices                   *
 *-----------------------------------------------------------*
 * The following enumeration gives a symbolic name to each   *
 * IDASPGMR optional output. The symbolic names are used as  *
 * indices into the iopt and ropt arrays passed to IDAMalloc.*
 * The IDASPGMR optional outputs are:                        *
 *                                                           *
 * iopt[SPGMR_NPE]  : number of preconditioner evaluations, i.e.  *
 *                    of calls made to user's precond function.   *
 *                                                           *
 * iopt[SPGMR_NLI]  : number of linear iterations.           *
 *                                                           *
 * iopt[SPGMR_NPS]  : number of calls made to user's psolve  *
 *                    function.                              *
 *                                                           *
 * iopt[SPGMR_NCFL] : number of linear convergence failures. *
 *                                                           *
 * iopt[SPGMR_LRW]  : size (in real words) of real workspace *
 *                    matrices and vectors used by this module.   *
 *                                                           *
 * iopt[SPGMR_LIW]  : size (in integer words) of integer     *
 *                    workspace vectors used by this module. *
 *                                                           *
 *************************************************************/
```

4.5. Use by a C++ Application. IDA has been written in so that it permits use by applications written in C++ as well as in C. For this purpose, each IDA header file is wrapped with conditionally compiled lines reading `extern "C" { ... }`, conditional on the variable `__cplusplus` being defined. This directive causes the C++ compiler to use C-style names when compiling the function prototypes encountered. Users with C++ applications should also note that we have defined a boolean variable type, `boole`, since C has no such type. This name was chosen to avoid a conflict with the C++ type `bool`.

4.6. Data Types real, integer, boole. As part of the IDA package, the `llnltyps.h` file contains the definitions of the data types `real`, `integer`, and `boole`. IDA uses the type `real` for all floating point data, and the type `integer` for all integers related to the problem size N, such as N itself, the half-bandwidths in the IDABAND solver, and the integers stored in the length-N pivot arrays in both the IDADENSE and IDABAND solvers. These types make it easy to have IDA solve problems of virtually any size using single or double precision arithmetic. The type `real` can be `double` or `float` and the type `integer` can be `int` or `long int`. The default settings are `double` and `int`. The type `boole`, which is equated to type `int`, was added for convenience in working with boolean logic.

The file `llnltyps.h` also defines constants which allow IDA to branch on the setting for types `real` and `integer` at compile time. Within IDA, this ability is needed in two places. One is for the macro `RCONST`, by which real constants are set. The other is in setting the correct data type in MPI calls for reduction operations.

The user's program can still use the type `double` or `float` for arguments of type `real`, and `int` or `long int` instead of `integer`, provided that the choice of type there matches the typedefs for `real` and `integer` in the file `llnltyps.h`.

4.6.1. Changing type real. The user can change the precision of IDA arithmetic from double to single by changing the typedef

```
typedef double real;          to          typedef float real;
```

in `llnltyps.h`, and by changing the constant definitions

```
#define LLNL_FLOAT  0         to          #define LLNL_FLOAT  1
#define LLNL_DOUBLE 1                     #define LLNL_DOUBLE 0
```

Changing from double precision to single precision arithmetic also requires minor changes in the implementation file `llnlmath.c` for the LLNLMATH module of IDA. The `RPowerR` and `RSqrt` functions compute a real number raised to a real power and the square root of a number, respectively. The default implementation of these routines calls standard C math library functions which do double precision arithmetic. If the user wants IDA to perform only single precision arithmetic, these implementations should be changed to call single precision routines which are available on the user's machine.

4.6.2. Changing type integer. IDA uses the type `integer` for all quantities related to problem size. On some machines the size of an `int` and a `long int` are the same, but this is not always the case. The size `int` may be too small on a machine for a very large problem. In this case, the user should change the typedef

```
typedef int integer;        to        typedef long int integer;
```

in `llnltyps.h`, and change the constant definitions

```
#define LLNL_INT  1         to        #define LLNL_INT  0
#define LLNL_LONG 0                   #define LLNL_LONG 1
```

4.6.3. Type boole. In order to support the use of boolean variables, a type `boole` has been added in `llnltyps.h`. This type is simply equated to type `int`. In addition, the constants `FALSE` (equal to 0) and `TRUE` (equal to 1) are defined.

5. Providing Alternate Linear Solver Modules. The central IDA module interfaces with the linear solver module to be used by way of calls to five routines. These are denoted here by `linit`, `lsetup`, `lsolve`, `lperf`, and `lfree`. Briefly, their purposes are as follows:

- `linit`: initialize and allocate memory specific to the linear solver;
- `lsetup`: evaluate and preprocess the Jacobian or preconditioner;
- `lsolve`: solve the linear system;
- `lperf`: monitor performance and issue warnings;
- `lfree`: free the linear solver memory.

The `lperf` routine is intended only for use in Krylov method modules, for which poor performance of the iterative method can be detected and reported to the user.

These routines necessarily have fixed call sequences. Thus a user wishing to implement another linear solver within the IDA package must adhere to this set of interfaces.

The following is a complete description of the call list for each of these routines. Note that the call list of each routine includes a pointer to the main IDA memory block, by which the routine can access various data related to the IDA solution. The contents of this memory block are given in the file `ida.h` (but not reproduced here, for the sake of space).

```
/*****************************************************************
 *                                                               *
 * int (*ida_linit)(IDAMem IDA_mem, boole *setupNonNull);        *
 *---------------------------------------------------------------*
 * The purpose of ida_linit is to allocate memory for the        *
 * solver-specific fields in the structure *(idamem->ida_lmem) and *
 * perform any needed initializations of solver-specific memory, *
 * such as counters/statistics. The ida_linit routine should set *
 * *setupNonNull to be TRUE if the setup operation for the linear *
 * solver is non-empty and FALSE if the setup operation does     *
 * nothing. An (*ida_linit) should return LINIT_OK (== 0) if it has*
 * successfully initialized the IDA linear solver and LINIT_ERR  *
 * (== -1) otherwise. These constants are defined above. If an   *
 * error does occur, an appropriate message should be sent to    *
 * (idamem->errfp).                                              *
 *                                                               *
 *****************************************************************/

/*****************************************************************
 *                                                               *
 * int (*ida_lsetup)(IDAMem IDA_mem, N_Vector yyp, N_Vector ypp, *
 *                   N_Vector resp,                              *
 *            N_Vector tempv1, N_Vector tempv2, N_Vector tempv3); *
 *---------------------------------------------------------------*
 * The job of ida_lsetup is to prepare the linear solver for     *
 * subsequent calls to ida_lsolve. Its parameters are as follows: *
 *                                                               *
 * idamem - problem memory pointer of type IDAMem. See the big   *
 *          typedef earlier in this file.                        *
 *                                                               *
 *                                                               *
 * yyp   - the predicted y vector for the current IDA internal   *
 *         step.                                                 *
 *                                                               *
 * ypp   - the predicted y' vector for the current IDA internal  *
 *         step.                                                 *
 *                                                               *
 * resp  - F(tn, yyp, ypp).                                      *
```

```
 *                                                                  *
 * tempv1, tempv2, tempv3 - temporary N_Vectors provided for use    *
 *         by ida_lsetup.                                           *
 *                                                                  *
 * The ida_lsetup routine should return SUCCESS (=0) if successful,*
 * the positive value LSETUP_ERROR_RECVR for a recoverable error,   *
 * and the negative value LSETUP_ERROR_NONRECVR for an              *
 * unrecoverable error.  The code should include the file ida.h .   *
 *                                                                  *
 ********************************************************************/

/********************************************************************
 *                                                                  *
 * int (*ida_lsolve)(IDAMem IDA_mem, N_Vector b, N_Vector ycur,     *
 *                 NPVector ypcur, N_Vector rescur);                *
 *------------------------------------------------------------------*
 * ida_lsolve must solve the linear equation P x = b, where         *
 * P is some approximation to the system Jacobian                   *
 *                  J = (dF/dy) + cj (dF/dy')                       *
 * evaluated at (tn,ycur,ypcur) and the RHS vector b is input.      *
 * The N-vector ycur contains the solver's current approximation    *
 * to y(tn), ypcur contains that for y'(tn), and the vector rescur  *
 * contains the N-vector residual F(tn,ycur,ypcur).                 *
 * The solution is to be returned in the vector b. ida_lsolve       *
 * returns the positive value LSOLVE_ERROR_RECVR for a              *
 * recoverable error and the negative value LSOLVE_ERROR_NONRECVR   *
 * for an unrecoverable error. Success is indicated by a return     *
 * value SUCCESS = 0. The code should include the file ida.h .      *
 *                                                                  *
 ********************************************************************/

/********************************************************************
 *                                                                  *
 * int (*ida_lperf)(IDAMem IDA_mem, int perftask);                  *
 *                                                                  *
 *------------------------------------------------------------------*
 * ida_lperf is called two places in IDA where linear solver        *
 * performance data is required by IDA. For perftask = 0, an        *
 * initialization of performance variables is performed, while for  *
 * perftask = 1, the performance is evaluated.                      *
 *                                                                  *
 ********************************************************************/

/********************************************************************
 *                                                                  *
 * int (*ida_lfree)(IDAMem IDA_mem);                                *
 *------------------------------------------------------------------*
 * ida_lfree should free up any memory allocated by the linear      *
```

```
 * solver. This routine is called once a problem has been          *
 * completed and the linear solver is no longer needed.            *
 *                                                                 *
 *******************************************************************/
```

6. A Band-Block-Diagonal Preconditioner Module: `IDABBDPRE`. A principal reason for using a DAE solver such as IDA lies in the solution of partial differential equations (PDEs). Moreover, the use of a Krylov iterative method for the solution of many such problems is motivated by the nature of the underlying system of linear equations (4) that must be solved at each time step. The linear algebraic system is large, sparse, and often structured. However, if a Krylov iterative method is to be effective in this setting, then a nontrivial preconditioner needs to be used. Otherwise, the rate of convergence of the Krylov iterative method is usually unacceptably slow. Unfortunately, effective preconditioners tend to be problem-specific.

However, we have developed one type of preconditioner that treats a rather broad class of PDE-based problems. It applies when the parallel version of IDA is used with `IDASPGMR` as the linear solver. This software module, `IDABBDPRE`, is included with the IDA package. A given time-dependent PDE system may be discretized in space by any method, giving a semi-discrete system, which is a DAE system in time. However, this system is assumed to have predominantly local coupling, and the ordering of the variables on each processor is assumed to reflect that local coupling. This module then generates a preconditioner that is a block-diagonal matrix with each block being a band matrix. The blocks need not have the same number of super- and sub-diagonals and these numbers may vary from block to block. This Band-Block-Diagonal Preconditioner module is called `IDABBDPRE`.

6.1. The algorithm. To obtain these preconditioners, think of the spatial domain of the computational PDE problem as being subdivided into M non-overlapping subdomains. Each of these subdomains is then assigned to one of the M processors to be used to solve the DAE system in parallel. The basic idea is to isolate the preconditioning so that it is local to each processor, and also to use a (possibly cheaper) approximate residual function. This requires the definition of a new function $G(t, y, y')$ which approximates the function $F(t, y, y')$ in the definition of the DAE system (1). The choice $G = F$ is certainly allowed, but a less expensive choice may be just as effective for preconditioning. Corresponding to the domain decomposition (and distribution of the system over the processors), there is a decomposition of the solution vectors y and y' into M disjoint blocks y_m and y'_m, and a decomposition of G into blocks G_m. As computed on processor m, the block G_m depends on (y_m, y'_m) and also on components of blocks (y_ℓ, y'_ℓ) associated with neighboring subdomains (so-called ghost-cell data). Let $\bar{y}_m$ and $\bar{y}'_m$ denote, respectively, y_m and y'_m augmented with those other components on which G_m depends. Then we have

$$G(t, y, y') = [G_1(t, \bar{y}_1, \bar{y}'_1), G_2(t, \bar{y}_2, \bar{y}'_2), \ldots, G_M(t, \bar{y}_M, \bar{y}'_M)]^T$$

and each of the blocks $G_m(t, \bar{y}_m, \bar{y}'_m)$ is uncoupled from the others.

The preconditioner associated with this decomposition is

$$P = diag[P_1, P_2, \ldots, P_M]$$

where P_m is a difference quotient approximation to

$$J_m = \frac{\partial G_m}{\partial y_m} + \alpha \frac{\partial G_m}{\partial y'_m} \ .$$

This matrix is taken to be banded, with upper and lower half-bandwidths `mu` and `ml`, defined as the number of non-zero diagonals above and below the main diagonal, respectively.

However, the true band structure of J_m may well be larger than that given by `mu` and `ml`. For this reason, another pair of half-bandwidths, `mudq` and `mldq`, is specified for use in the difference quotient approximation procedure. Thus P_m is computed as a matrix with bandwidth `mudq + mldq + 1`, using `mudq + mldq + 2` evaluations of G_m, but only a matrix of bandwidth `mu + ml + 1` is retained. Neither of these pairs, (`mu, ml`) or (`mudq, mldq`), need be the true values of the half-bandwidths of J_m, if smaller values provide a more efficient preconditioner. Also, they need not be the same on every processor.

To carry out this idea, the communication of the ghost-cell data required by G_m, from neighboring processors ℓ to processor m, is isolated into a separate operation. Once this communication is done at a given point (t, y, y') where P is being computed, the evaluations of G_m needed to generate the difference quotients for P_m involve *no* communication. Then the solution of the complete preconditioner linear system $Px = b$ obviously reduces to solving each of the equations

$$P_m x_m = b_m \ , \tag{9}$$

and this is done by a banded LU factorization of P_m followed by a banded backsolve. Both of those operations can be performed completely in parallel for $m = 1, \ldots M$.

Though intended for parallel usage, this module can be used in the case of a single processor, where it generates a banded approximate Jacobian as the preconditioner.

Similar block-diagonal preconditioners could be considered with a different treatment of the blocks J_m. For example, incomplete LU factorization or an iterative method could be used instead of banded LU factorization.

6.2. Using `IDABBDPRE`. To use this `IDABBDPRE` module, the user must supply the following two functions, which the module calls to construct P, in addition to the user-supplied residual function `resfn`:

- A function `glocal (tt, yy, yp, gg, rdata)` must be supplied by the user to compute $G(t, y, y')$. It loads the vector `gg` as a function of `tt`, `yy`, and `yp`. Although `yy, yp,` and `gg` are all of type `N_Vector`, only the local segment of each, of length `Nlocal`, is to be accessed in the routine `glocal`.
- A function `gcomm (yy, yp, rdata)` must be supplied to perform all inter-processor communications necessary for the execution of the `glocal` function, i.e. communication of components of the input vectors `yy` and `yp`, of type `N_Vector`.

Both of these functions receive as input the same pointer `rdata` (to user problem data) as that passed by the user to `IDAMalloc` and passed to the user's `resfn`. Both are to return an `int` equal to 0 (indicating success), or else 1 or -1 (indicating recoverable or non-recoverable failure, respectively), just as for `resfn`. The user is responsible for providing

space, presumably within (*rdata), for data (the ghost-cell data) that is communicated by gcomm from the other processors, in a manner suitable for use by glocal, which is not expected to do any communication.

Each call to gcomm is preceded by a call to resfn with the same arguments yy and yp. Thus gcomm can omit any communications done by resfn if these are relevant to the execution of the local function glocal.

The user's calling program should #include the file idabbdpre.h, and in place of the call to IDASpgmr it should include the following two calls:

- IBBDData pdata;
  pdata = IBBDAlloc(Nlocal, mudq,mldq, mu,ml, dq_rel_y, glocal, gcomm,
  idamem, rdata);
  if (pdata == NULL) return(1);
- IDASpgmr(idamem, IBBDPrecon, IBBDPSol, gstype, ..., pdata);

The names IBBDPrecon, IBBDPSol in the call to IDASpgmr are not dummy names, but refer to the specific routines in the IDABBDPRE module. After solving the problem, along with the other memory-freeing calls, the user program should include the following:

- IBBDFree(pdata); to free the IDABBDPRE memory block.

A detailed description of the user interface to IBBDAlloc is given in the following (excerpted from idabbdpre.h).

```
IBBDData IBBDAlloc(integer Nlocal, integer mudq, integer mldq,
                   integer mukeep, integer mlkeep, real dq_rel_yy,
                   IDALocalFn glocal, IDACommFn gcomm,
                   void *idamem, void *res_data);

/*****************************************************************
 * Function : IBBDAlloc                                          *
 *---------------------------------------------------------------*
 * IBBDAlloc allocates and initializes an IBBDData structure     *
 * to be passed to IDASpgmr (and subsequently used by IBBDPrecon *
 * and IBBDPSol.                                                 *
 *                                                               *
 * The parameters of IBBDAlloc are as follows:                   *
 *                                                               *
 * Nlocal  is the length of the local block of the vectors yy etc.*
 *         on the current processor.                             *
 *                                                               *
 * mudq, mldq  are the upper and lower half-bandwidths to be used *
 *         in the computation of the local Jacobian blocks.      *
 *                                                               *
 * mukeep, mlkeep are the upper and lower half-bandwidths to be  *
 *         used in saving the Jacobian elements in the local     *
 *         block of the preconditioner matrix PP.                *
```

```
*                                                               *
* dq_rel_yy is an optional input.  It is the relative increment  *
*         to be used in the difference quotient routine for      *
*         Jacobian calculation in the preconditioner.  The       *
*         default is sqrt(unit roundoff), and specified by       *
*         passing dq_rel_yy = 0.                                 *
*                                                               *
* glocal    is the name of the user-supplied function G(t,y,y')  *
*         that approximates F and whose local Jacobian blocks    *
*         are to form the preconditioner.                        *
*                                                               *
* gcomm   is the name of the user-defined function that performs *
*         necessary inter-processor communication for the        *
*         execution of glocal.                                   *
*                                                               *
* idamem  is the pointer to the IDA memory returned by IDAMalloc.*
*                                                               *
* res_data is a pointer to the optional user data block, as      *
*         passed to IDAMalloc.                                   *
*                                                               *
* IBBDAlloc returns the storage allocated (type IBBDData),       *
* or NULL if the request for storage cannot be satisfied.        *
*****************************************************************/
```

Three optional outputs associated with this module are available by way of macros. These are:

- `IBBD_RPWSIZE(pdata)` = size of the real workspace (local to the current processor) used by `IDABBDPRE`.
- `IBBD_IPWSIZE(pdata)` = size of the integer workspace (local to the current processor) used by `IDABBDPRE`.
- `IBBD_NGE(pdata)` = cumulative number of G evaluations (calls to `glocal`) so far.

The costs associated with using `IDABBDPRE` also include `nsetups` LU factorizations, `nsetups` calls to `gcomm`, and `nps` banded backsolve calls, where `nsetups` and `nps` are the IDA optional outputs `iopt[NSETUPS]` and `iopt[SPGMR_NPS]`.

7. Example Problems. The IDA package includes eight example programs. These are based on three DAE system problems, two of which are solved in several different ways. The last of those two, a food web problem, is the most difficult and most realistic. Collectively, the examples are intended to illustrate the usage of both the serial and parallel versions of IDA, the usage of all three linear system modules, the use of the `IDACalcIC` routine, and the use of the `IDABBDPRE` preconditioner module. In the following, we present the three DAE problems, and describe how each is solved with IDA.

7.1. Robertson Kinetics Problem. This example, due to Robertson, 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:

$$\begin{cases} dy^1/dt = -.04y^1 + 10^4 y^2 y^3 \\ dy^2/dt = +.04y^1 - 10^4 y^2 y^3 - 3 \cdot 10^7 (y^2)^2 \\ 0 = y^1 + y^2 + y^3 - 1 \end{cases} \tag{10}$$

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

This problem was solved only in one serial case using IDADENSE, the simplest linear solver supplied with IDA. It illustrates the application of IDADENSE, with a user-supplied Jacobian function, for those problems to which a dense solver is applicable.

The code and corresponding output can be found as `robx.c` and `robx.output` in the distributed package.

7.2. Heat Equation Problem. 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 equations solved are:

$$\begin{cases} \partial u/\partial t = u_{xx} + u_{yy} & \text{(interior)} \\ u = 0. & \text{(boundary).} \end{cases} \tag{11}$$

Initial conditions are given by $u = 16x(1-x)y(1-y)$, where the spatial domain is the unit square $0 \le x, y \le 1$, and the time interval is $0 \le t \le 10.24$.

We discretize this PDE system (11) (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 $u^i(x_j, y_k, t)$ grouped first by x, and then by y. At each spatial boundary point, the boundary condition is coupled algebraically into the adjacent interior points by the central differencing scheme.

This problem was solved in four different ways, with the following example programs:

- `heatsb`: serial version of IDA, band linear solver. The half-bandwidths are 10.
- `heatsk`: serial version of IDA, Krylov (GMRES) linear solver with a user-supplied preconditioner. As a preconditioner, we use the diagonal elements of the matrix J.
- `heatpk`: parallel version of IDA, Krylov (SPGMR) linear solver with a user-supplied preconditioner. We use a 5×5 subgrid on each of 4 processors. For the preconditioner, we again use the diagonal elements of the matrix J.
- `heatbbd`: parallel version of IDA, Krylov (SPGMR) linear solver with `IDABBDPRE` preconditioner module. We use a 5×5 subgrid on each of 4 processors. We use half-bandwidths `mudq = mldq =` 5 on each processor for the difference quotient scheme, but keep only a tridiagonal matrix (`mu = ml =` 1).

The source program for all four cases, along with the corresponding output files are available in the distributed package. They are not included in this document.

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

$$\left\{ \begin{array}{rl} \partial c^i/\partial t = R_i(x,y,c) + d_i(c^i_{xx} + c^i_{yy}) & (i = 1, 2, \ldots, p) \ , \\ 0 = R_i(x,y,c) + d_i(c^i_{xx} + c^i_{yy}) & (i = p+1, \ldots, s) \ , \end{array} \right. \tag{12}$$

with

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

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:

$$\left\{ \begin{array}{l} a_{ii} = -1 \quad (\text{all } i) \\ a_{ij} = -0.5 \cdot 10^{-6} \quad (i \le p, j > p) \\ a_{ij} = 10^4 \ (i > p, j \le p) \\ (\text{all other } a_{ij} = 0) \ , \end{array} \right.$$

$$\left\{ \begin{array}{l} b_i = b_i(x,y) = (1 + \alpha xy + \beta \sin(4\pi x) \sin(4\pi y)) \quad (i \le p) \\ b_i = b_i(x,y) = -(1 + \alpha xy + \beta \sin(4\pi x) \sin(4\pi y)) \quad (i > p) \ , \end{array} \right.$$

and

$$\left\{ \begin{array}{l} d_i = 1 \quad (i \le p) \\ d_i = 0.5 \quad (i > p). \end{array} \right.$$

The spatial domain is the unit square $0 \le x, y \le 1$, and the time interval is $0 \le t \le 1$. The boundary conditions are of Neumann type (zero normal derivatives) everywhere. The coefficients are such that a unique stable equilibrium is guaranteed to exist when $\alpha = \beta = 0$ [2]. Empirically, a stable equilibrium appears to exist for (12) 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:

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

We discretize this PDE system (12) (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 $L = 20$. See also [4], where various cases of this problem are solved with DASPK.

This problem was solved in three different ways, with the following three example programs:

- `webpk`: parallel version of IDA, Krylov (SPGMR) linear solver with a user-supplied preconditioner. We use a $Lsub \times Lsub$ subgrid, with $Lsub = 10$, on each of 4 processors. For the preconditioner, we take the block-diagonal matrix with 2×2 blocks arising from the reaction coefficients $\partial R_i/\partial c$ only.
- `webbbd`: parallel version of IDA, Krylov (SPGMR) linear solver with `IDABBDPRE` preconditioner module. We use half-bandwidths `mudq = mldq =` $s \cdot Lsub = 20$ for the difference quotient scheme, but retain only a matrix with bandwidth 5 by setting `mu = ml =` 2.

In all three cases, the flat predator initial values are not consistent with the quasi-steady equations for the predator species, and so we call `IDACalcIC` to correct those values. In the two parallel programs, we use a logically square array of processors and corresponding Cartesian subdomain decomposition. The source program for the second case, `webpk.c`, is given in its entirety in the Appendix. The output for this case is also included there.

8. Availability. The IDA package is being released for general distribution at this time. Interested potential users should contact Alan Hindmarsh (alanh@llnl.gov) or Allan Taylor (agtaylor@llnl.gov).

REFERENCES

[1] K. E. Brenan, S. L. Campbell and L. R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, second edition, SIAM, 1996.

[2] Peter N. Brown, *Decay to Uniform States in Food Webs*, SIAM J. Appl. Math., 46 (1986), pp. 376–392.

[3] Peter N. Brown and Alan C. Hindmarsh, *Reduced Storage Matrix Methods in Stiff ODE Systems*, J. Appl. Math. & Comp. 31 (1989), pp. 40–91.

[4] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold *Using Krylov Methods in the Solution of Large-Scale Differential-Algebraic Systems*, SIAM J. Sci. Comput., 15 (1994), pp. 1467–1488.

[5] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold, *Consistent Initial Condition Calculation for Differential-Algebraic Systems*, SIAM J. Sci. Comput., 19 (1998), pp. 1495–1512.

[6] Peter N. Brown and Yousef Saad, *Hybrid Krylov Methods for Nonlinear Systems of Equations*, SIAM J. Sci. Stat. Comput., 11 (1990), pp. 450–481.

[7] George D. Byrne and Alan C. Hindmarsh, *User Documentation for PVODE, An ODE Solver for Parallel Computers*, Lawrence Livermore National Laboratory report UCRL-ID-130884, May 1998.

[8] George D. Byrne and Alan C. Hindmarsh, *PVODE, An ODE Solver for Parallel Computers*, Int. J. High Perf. Comput. Applic., 13, No. 4 (1999), pp. 354–365.

[9] Scott D. Cohen and Alan C. Hindmarsh, *CVODE User Guide*, Lawrence Livermore National Laboratory report UCRL-MA-118618, September 1994.

[10] Scott D. Cohen and Alan C. Hindmarsh, *CVODE, a Stiff/Nonstiff ODE Solver in C*, Computers in Physics, 10, No. 2 (1996), pp. 138–143.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum, *Using MPI Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, Cambridge, MA, 1994.

[12] Yousef Saad and Martin H. Schultz, *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems*, SIAM J. Sci. Stat. Comp. 7 (1986), pp. 856–869.

[13] Allan G. Taylor and Alan C. Hindmarsh, *User Documentation for KINSOL, A Nonlinear Solver for Sequential and Parallel Computers*, Lawrence Livermore National Laboratory report UCRL-ID-131185, July 1998.

9. Appendix: Listing and Output of Predator-Prey Example Program.

```
/*************************************************************************
 * File:       webpk.c                                                   *
 * Written by: Allan G. Taylor and Alan C. Hindmarsh @ LLNL              *
 * Version of: 7 February 2000                                           *
 *-----------------------------------------------------------------------*
 *
 * Example program for IDA: Food web, parallel, GMRES, user preconditioner.
 *
 * This example program for IDA uses IDASPGMR as the linear solver.
 * It is written for a parallel computer system and uses a block-diagonal
 * preconditioner (setup and solve routines) for the IDASPGMR package.
 * It was originally run on a Sun SPARC cluster and used MPICH.
 *
 * The mathematical problem solved in this example is a DAE system that
 * arises from a system of partial differential equations after spatial
 * discretization.  The PDE system is a food web population model, with
 * predator-prey interaction and diffusion on the unit square in two
 * dimensions. The dependent variable vector is:
 *
 *       1   2         ns
 * c = (c , c ,  ..., c  ) ,   ns = 2 * np
 *
 * and the PDE's are as follows:
 *
 *     i             i      i
 *   dc /dt = d(i)*(c    + c  )  +  R (x,y,c)    (i=1,...,np)
 *                   xx     yy       i
 *
 *
 *                   i      i
 *  0       = d(i)*(c    + c  )  +  R  (x,y,c)   (i=np+1,...,ns)
 *                   xx     yy       i
 *
 *   where the reaction terms R are:
 *
 *                   i             ns         j
 *   R  (x,y,c)  =  c  * (b(i)  + sum a(i,j)*c )
 *    i                           j=1
 *
 * The number of species is ns = 2 * np, with the first np being prey and
 * the last np being predators. The coefficients a(i,j), b(i), d(i) are:
 *
 *   a(i,i) = -AA  (all i)
 *   a(i,j) = -GG  (i <= np , j >  np)
 *   a(i,j) =  EE  (i >  np,  j <= np)
 *   all other a(i,j) = 0
```

```
 *   b(i) = BB*(1+ alpha * x*y + beta*sin(4 pi x)*sin(4 pi y))  (i <= np)
 *   b(i) =-BB*(1+ alpha * x*y + beta*sin(4 pi x)*sin(4 pi y))  (i  > np)
 *   d(i) = DPREY  (i <= np)
 *   d(i) = DPRED  (i > np)
 *
 * NOTE: The above equations are written in 1-based indices, whereas the
 * code has 0-based indices, being written in C.
 *
 * The various scalar parameters required are set using 'define' statements
 * or directly in routine InitUserData.  In this program, np = 1, ns = 2.
 * The boundary conditions are homogeneous Neumann: normal derivative  =  0.
 *
 * A polynomial in x and y is used to set the initial values of the first
 * np variables (the prey variables) at each x,y location, while initial
 * values for the remaining (predator) variables are set to a flat value,
 * which is corrected by IDACalcIC.
 *
 * The PDEs are discretized by central differencing on a MX by MY mesh,
 * and so the system size Neq is the product MX * MY * NUM_SPECIES.
 * The system is actually implemented on submeshes, processor by processor,
 * with an MXSUB by MYSUB mesh on each of NPEX * NPEY processors.
 *
 * The DAE system is solved by IDA using the IDASPGMR linear solver, which
 * uses the preconditioned GMRES iterative method to solve linear systems.
 * The precondtioner supplied to IDASPGMR is the block-diagonal part of
 * the Jacobian with ns by ns blocks arising from the reaction terms only.
 * Output is printed at t = 0, .001, .01, .1, .4, .7, 1.
 *
 * References:
 * [1] Peter N. Brown and Alan C. Hindmarsh,
 *     Reduced Storage Matrix Methods in Stiff ODE systems,
 *     Journal of Applied Mathematics and Computation, Vol. 31 (May 1989),
 *     pp. 40-91.
 *
 * [2] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold,
 *     Using Krylov Methods in the Solution of Large-Scale Differential-
 *     Algebraic Systems, SIAM J. Sci. Comput., 15 (1994), pp. 1467-1488.
 *
 * [3] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold,
 *     Consistent Initial Condition Calculation for Differential-
 *     Algebraic Systems, SIAM J. Sci. Comput., 19 (1998), pp. 1495-1512.
 *
 ***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "llnltyps.h"   /* Definitions of real, integer, boole, TRUE, FALSE.*/
```

```
#include "iterativ.h"   /* Contains the enum for types of preconditioning.  */
#include "ida.h"        /* Main IDA header file.                            */
#include "idaspgmr.h"   /* Use IDASPGMR linear solver.                      */
#include "nvector.h"    /* Definitions of type N_Vector, macro N_VDATA.     */
#include "llnlmath.h"   /* Contains RSqrt and UnitRoundoff routines.        */
#include "smalldense.h" /* Contains definitions for denalloc routine.       */
#include "mpi.h"        /* MPI library routines.                            */

/* Problem Constants. */

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

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

#define MXSUB       10    /* Number of x mesh points per processor subgrid */
#define MYSUB       10    /* Number of y mesh points per processor subgrid */
#define NPEX        2     /* Number of subgrids in the x direction */
#define NPEY        2     /* Number of subgrids in the y direction */
#define MX          (MXSUB*NPEX)      /* MX = number of x mesh points */
#define MY          (MYSUB*NPEY)      /* MY = number of y mesh points */
#define NSMXSUB     (NUM_SPECIES * MXSUB)
#define NEQ         (NUM_SPECIES*MX*MY) /* Number of equations in system */
#define AA          RCONST(1.0)    /* Coefficient in above eqns. for a */
#define EE          RCONST(10000.) /* Coefficient in above eqns. for a */
#define GG          RCONST(0.5e-6) /* Coefficient in above eqns. for a */
#define BB          RCONST(1.0)    /* Coefficient in above eqns. for b */
#define DPREY       RCONST(1.0)    /* Coefficient in above eqns. for d */
#define DPRED       RCONST(0.05)   /* Coefficient in above eqns. for d */
#define ALPHA       RCONST(50.)    /* Coefficient alpha in above eqns. */
#define BETA        RCONST(1000.)  /* Coefficient beta in above eqns. */
#define AX          RCONST(1.0)    /* Total range of x variable */
#define AY          RCONST(1.0)    /* Total range of y variable */
#define RTOL        RCONST(1.e-5)  /*  rtol tolerance */
#define ATOL        RCONST(1.e-5)  /*  atol tolerance */
#define ZERO        RCONST(0.)     /* 0. */
#define ONE         RCONST(1.0)    /* 1. */
#define NOUT        6
#define TMULT       RCONST(10.0)   /* Multiplier for tout values */
#define TADD        RCONST(0.3)    /* Increment for tout values */


/* User-defined vector accessor macro IJ_Vptr. */

/* IJ_Vptr is defined in order to express the underlying 3-d structure of the
   dependent variable vector from its underlying 1-d storage (an N_Vector).
   IJ_Vptr(vv,i,j) returns a pointer to the location in vv corresponding to
```

```
   species index is = 0, x-index ix = i, and y-index jy = j.                */

#define IJ_Vptr(vv,i,j)   (&(((vv)->data)[(i)*NUM_SPECIES + (j)*NSMXSUB]))


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

typedef struct {
  integer Neq, ns, np, thispe, npes, ixsub, jysub, npex, npey,
          mxsub, mysub, nsmxsub, nsmxsub2;
  real dx, dy, **acoef;
  real cox[NUM_SPECIES], coy[NUM_SPECIES], bcoef[NUM_SPECIES],
       rhs[NUM_SPECIES], cext[(MXSUB+2)*(MYSUB+2)*NUM_SPECIES];
  MPI_Comm comm;
  N_Vector rates;
  real **PP[MXSUB][MYSUB];
  integer *pivot[MXSUB][MYSUB];
} *UserData;


/* Prototypes for private Helper Functions. */

static UserData AllocUserData(machEnvType machEnv);

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

static void FreeUserData(UserData webdata);

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

static void PrintOutput(long int iopt[], real ropt[], N_Vector cc, real time,
                        UserData webdata, MPI_Comm comm);

static void PrintFinalStats(long int iopt[]);

static int rescomm(N_Vector cc, N_Vector cp, void *rdata);

static void BSend(MPI_Comm comm, integer thispe, integer ixsub, integer jysub,
                  integer dsizex, integer dsizey, real carray[]);

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

static void BRecvWait(MPI_Request request[], integer ixsub, integer jysub,
```

```
                     integer dsizex, real cext[], real buffer[]);

static int reslocal(real tt, N_Vector cc, N_Vector cp, N_Vector res,
    void *rdata);

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

static real dotprod(integer size, real *x1, real *x2);


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

static int resweb(integer Neq, real time, N_Vector cc, N_Vector cp,
                  N_Vector resval, void *rdata);

static int Precondbd(integer Neq, real tt, N_Vector cc, N_Vector cp,
                     N_Vector rr, real cj, ResFn res,  void *rdata,
                     void *Pdata, N_Vector ewt, N_Vector constraints,
                     real hh, real uround, long int *nrePtr,
                     N_Vector tempv1, N_Vector tempv2, N_Vector tempv3);

static int PSolvebd(integer Neq, real tt, N_Vector cc, N_Vector cp,
                    N_Vector rr, real cj, ResFn res, void *rdata,
                    void *Pdata, N_Vector ewt, real delta,
                    N_Vector rvec, N_Vector zvec,
                    long int *nfePtr, N_Vector tempv);


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

main(int argc, char *argv[])
{
  integer SystemSize, thispe, npes, local_N;
  real rtol, atol, ropt[OPT_SIZE], t0, tout, tret;
  long int iopt[OPT_SIZE];
  N_Vector cc, cp, res, id;
  UserData webdata;
  int maxl, iout, flag, retval, i, itol, itask;
  boole optIn;
  void *mem;
  IDAMem idamem;
  MPI_Comm comm;
  machEnvType machEnv;

  /* Set communicator, and get processor number and total number of PE's. */

  MPI_Init(&argc, &argv);
```

```
comm = MPI_COMM_WORLD;
MPI_Comm_rank(comm, &thispe);
MPI_Comm_size(comm, &npes);

if (npes != NPEX*NPEY) {
  if (thispe == 0)
    printf("\n npes = %d not equal to NPEX*NPEY = %d\n", npes, NPEX*NPEY);
  return(1); }

/* Set local length (local_N) and global length (SystemSize). */

local_N = MXSUB*MYSUB*NUM_SPECIES;
SystemSize = NEQ;

/* Set machEnv block. */

machEnv = PVecInitMPI(comm, local_N, SystemSize, &argc, &argv);
if (machEnv == NULL) return(1);

/* Set up user data block webdata. */

webdata = AllocUserData(machEnv);
InitUserData(webdata, thispe, npes, comm);

/* Create needed vectors, and load initial values.
   The vector res is used temporarily only.        */

cc  = N_VNew(SystemSize, machEnv);
cp  = N_VNew(SystemSize, machEnv);
res = N_VNew(SystemSize, machEnv);
id  = N_VNew(SystemSize, machEnv);

SetInitialProfiles(cc, cp, id, res, webdata);

N_VFree(res);

/* Set remaining inputs to IDAMalloc. */

t0 = ZERO;
itol = SS; rtol = RTOL; atol = ATOL;
optIn = FALSE;

/* Call IDAMalloc to initialize IDA.
   First NULL argument  = constraints vector, not used here.
   Second NULL argument = file pointer for error messages (sent to stdout).
   A pointer to IDA problem memory is returned and stored in idamem.      */

mem = IDAMalloc(SystemSize, resweb, webdata, t0, cc, cp, itol,&rtol,&atol,
```

```
                id, NULL, NULL, optIn, iopt, ropt, machEnv);

if (mem == NULL) {
  if (thispe == 0) printf("IDAMalloc failed.");
  return(1); }
idamem = (IDAMem)mem;

/* Call IDASpgmr to specify the IDA linear solver IDASPGMR and specify
   the preconditioner routines supplied (Precondbd and PSolvebd).
   Optional input maxl (max. Krylov subspace dim.) is set to 10.   */

maxl = 10;
retval = IDASpgmr(idamem, Precondbd, PSolvebd, MODIFIED_GS,
                  maxl, 0, ZERO, ZERO, webdata);

if (retval != 0) {
  if (thispe == 0) printf("IDASpgmr call failed, returning %d \n",retval);
  return(1); }

/* Call IDACalcIC (with default options) to correct the initial values. */

tout = 0.001;
retval = IDACalcIC(idamem, CALC_YA_YDP_INIT, tout, ZERO, 0,0,0,0, ZERO);

if (retval != SUCCESS) {
  if (thispe == 0) printf("IDACalcIC failed. retval = %d\n",retval);
  return(1); }

/* On PE 0, print heading, basic parameters, initial values. */

if (thispe == 0) {
  printf("webpk: Predator-prey DAE parallel example problem for IDA \n\n");
  printf("Number of species ns: %d", NUM_SPECIES);
  printf("     Mesh dimensions: %d x %d", MX, MY);
  printf("     Total system size: %d\n",SystemSize);
  printf("Subgrid dimensions: %d x %d", MXSUB, MYSUB);
  printf("     Processor array: %d x %d\n", NPEX, NPEY);
  printf("Tolerance parameters:  rtol = %g   atol = %g\n", rtol, atol);
  printf("Linear solver: IDASPGMR     Max. Krylov dimension maxl: %d\n",
          maxl);
  printf("Preconditioner: block diagonal, block size ns,");
  printf(" via difference quotients \n\n");
}
PrintOutput(iopt, ropt, cc, t0, webdata, comm);

/* Loop over iout, call IDASolve (normal mode), print selected output. */

itask = NORMAL;
```

```
  for (iout = 1; iout <= NOUT; iout++) {

    flag = IDASolve(idamem, tout, t0, &tret, cc, cp, itask);

    if (flag != SUCCESS) {
      if (thispe == 0) printf("IDA failed, flag =%d.\n", flag);
      return(flag); }

    PrintOutput(iopt, ropt, cc, tret, webdata, comm);

    if (iout < 3) tout *= TMULT; else tout += TADD;

  } /* End of iout loop. */

  /* On PE 0, print final set of statistics. */

  if (thispe == 0) PrintFinalStats(iopt);

  /* Free memory. */

  N_VFree(cc); N_VFree(cp); N_VFree(id);
  IDAFree(idamem);
  FreeUserData(webdata);
  PVecFreeMPI(machEnv);
  MPI_Finalize();
  return(0);

} /* End of webpk main program. */


/********************** Private Helper Functions ***********************/


/*************************************************************************/
/* AllocUserData: Allocate memory for data structure of type UserData.   */

static UserData AllocUserData(machEnvType machEnv)
{
  integer ix, jy;
  UserData webdata;

  webdata = (UserData) malloc(sizeof *webdata);

  webdata->rates = N_VNew(NEQ, machEnv);

  for (ix = 0; ix < MXSUB; ix++) {
    for (jy = 0; jy < MYSUB; jy++) {
      (webdata->PP)[ix][jy] = denalloc(NUM_SPECIES);
```

```
      (webdata->pivot)[ix][jy] = denallocpiv(NUM_SPECIES);
    }
  }

  webdata->acoef = denalloc(NUM_SPECIES);

  return(webdata);

} /* End of AllocUserData. */


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

static void InitUserData(UserData webdata, integer thispe, integer npes,
                         MPI_Comm comm)
{
  int i, j, np;
  real *a1,*a2, *a3, *a4, *b, dx2, dy2, **acoef, *bcoef, *cox, *coy;

  webdata->jysub = thispe / NPEX;
  webdata->ixsub = thispe - (webdata->jysub)*NPEX;
  webdata->mxsub = MXSUB;
  webdata->mysub = MYSUB;
  webdata->npex = NPEX;
  webdata->npey = NPEY;
  webdata->ns = NUM_SPECIES;
  webdata->np = NPREY;
  webdata->dx = AX/(MX-1);
  webdata->dy = AY/(MY-1);
  webdata->Neq = NEQ;
  webdata->thispe = thispe;
  webdata->npes   = npes;
  webdata->nsmxsub = MXSUB * NUM_SPECIES;
  webdata->nsmxsub2 = (MXSUB+2)*NUM_SPECIES;
  webdata->comm = comm;

  /* Set up the coefficients a and b plus others found in the equations. */

  np = webdata->np;
  dx2 = (webdata->dx)*(webdata->dx); dy2 = (webdata->dy)*(webdata->dy);

  acoef = webdata->acoef;
  bcoef = webdata->bcoef;
  cox = webdata->cox;
  coy = webdata->coy;

  for (i = 0; i < np; i++) {
```

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

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

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

} /* End of InitUserData. */


/*********************************************************************/
/* FreeUserData: Free webdata memory.                                */

static void FreeUserData(UserData webdata)
{
  integer ix, jy;

  for (ix = 0; ix < MXSUB; ix++) {
    for (jy = 0; jy < MYSUB; jy++) {
      denfree((webdata->PP)[ix][jy]);
      denfreepiv((webdata->pivot)[ix][jy]);
    }
  }

  denfree(webdata->acoef);
  N_VFree(webdata->rates);
  free(webdata);

} /* End of FreeData. */


/*********************************************************************/
/* SetInitialProfiles: Set initial conditions in cc, cp, and id.
   A polynomial profile is used for the prey cc values, and a constant
```

```
   (1.0e5) is loaded as the initial guess for the predator cc values.
   The id values are set to 1 for the prey and 0 for the predators.
   The prey cp values are set according to the given system, and
   the predator cp values are set to zero.                          */

static void SetInitialProfiles(N_Vector cc, N_Vector cp, N_Vector id,
                               N_Vector res, UserData webdata)
{
  integer ixsub, jysub, mxsub, mysub, nsmxsub, np, ix, jy, is, loc, yloc;
  real *cxy, *idxy, *cpxy, dx, dy, xx, yy, xyfactor;

  ixsub = webdata->ixsub;
  jysub = webdata->jysub;
  mxsub = webdata->mxsub;
  mysub = webdata->mxsub;
  nsmxsub = webdata->nsmxsub;
  dx = webdata->dx;
  dy = webdata->dy;
  np = webdata->np;

  /* Loop over grid, load cc values and id values. */
  for (jy = 0; jy < mysub; jy++) {
    yy = (jy + jysub*mysub) * dy;
    for (ix = 0; ix < mxsub; ix++) {
      xx = (ix + ixsub*mxsub) * dx;
      xyfactor = 16.*xx*(1. - xx)*yy*(1. - yy);
      xyfactor *= xyfactor;

      cxy = IJ_Vptr(cc,ix,jy);
      idxy = IJ_Vptr(id,ix,jy);
      for (is = 0; is < NUM_SPECIES; is++) {
        if (is < np) { cxy[is] = 10. + (real)(is+1)*xyfactor; idxy[is] = ONE; }
else { cxy[is] = 1.0e5; idxy[is] = ZERO; }
      }
    }
  }

  /* Set c' for the prey by calling the residual function with cp = 0. */

  N_VConst(ZERO, cp);
  resweb(webdata->Neq, ZERO, cc, cp, res, webdata);
  N_VScale(-ONE, res, cp);

  /* Set c' for predators to 0. */

  for (jy = 0; jy < mysub; jy++) {
    for (ix = 0; ix < mxsub; ix++) {
      cpxy = IJ_Vptr(cp,ix,jy);
```

```
      for (is = np; is < NUM_SPECIES; is++) cpxy[is] = ZERO;
    }
  }
} /* End of SetInitialProfiles. */


/*****************************************************************/
/* PrintOutput: Print output values at output time t = tt.
   Selected run statistics are printed.  Then values of c1 and c2
   are printed for the bottom left and top right grid points only.
   (NOTE: This routine is specific to the case NUM_SPECIES = 2.)         */

static void PrintOutput(long int iopt[], real ropt[], N_Vector cc, real tt,
                        UserData webdata, MPI_Comm comm)
{
  int is;
  MPI_Status status;
  integer thispe, npelast, ilast, ix, jy;
  real *cdata, clast[2];

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

  /* Send c1 and c2 at top right mesh point from PE npes-1 to PE 0. */
  if (thispe == npelast) {
    ilast = NUM_SPECIES*MXSUB*MYSUB - 2;
    if (npelast != 0)
      MPI_Send(&cdata[ilast], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
    else { clast[0] = cdata[ilast]; clast[1] = cdata[ilast+1]; }
  }

  /* On PE 0, receive c1 and c2 at top right from PE npes - 1.
     Then print performance data and sampled solution values. */

  if (thispe == 0) {

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

    printf("\nTIME t = %e.     NST = %d,  k = %d,  h = %e\n",
           tt, iopt[NST], iopt[KUSED], ropt[HUSED]);
    printf("NRE = %d,  NNI = %d,  NLI = %d,  NPE = %d,  NPS = %d\n",  iopt[NRE],
           iopt[NNI], iopt[SPGMR_NLI], iopt[SPGMR_NPE], iopt[SPGMR_NPS]);

    printf("At bottom left:  c1, c2 = %e %e \n",   cdata[0], cdata[1]);
    printf("At top right:    c1, c2 = %e %e \n\n", clast[0], clast[1]);
  }
```

```
} /* End of PrintOutput. */


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

static void PrintFinalStats(long int iopt[])
{
  printf("\nFinal statistics: \n\n");
  printf("NST  = %5ld     NRE  = %5ld\n", iopt[NST], iopt[NRE]);
  printf("NNI  = %5ld     NLI  = %5ld\n", iopt[NNI], iopt[SPGMR_NLI]);
  printf("NPE  = %5ld     NPS  = %5ld\n", iopt[SPGMR_NPE],iopt[SPGMR_NPS]);
  printf("NETF = %5ld     NCFN = %5ld     NCFL = %5ld\n",
         iopt[NETF], iopt[NCFN], iopt[SPGMR_NCFL]);

} /* End of PrintFinalStats. */


/********** Functions Called by IDA, and supporting functions  **********/


/*********************************************************************/
/* resweb: System residual function for predator-prey system.
   To compute the residual function F, this routine calls:
      rescomm, for needed communication, and then
      reslocal, for computation of the residuals on this processor.  */

static int resweb(integer Neq, real tt, N_Vector cc, N_Vector cp,
                  N_Vector res,  void *rdata)
{
  int retval;
  UserData webdata;

  webdata = (UserData)rdata;

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

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

  return(0);

} /* End of resweb. */


/*********************************************************************/
/* rescomm: Communication routine in support of resweb.
```

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

static int rescomm(N_Vector cc, N_Vector cp, void *rdata)
{

  UserData webdata;
  real *cdata, *cext, buffer[2*NUM_SPECIES*MYSUB];
  integer thispe, ixsub, jysub, mxsub, mysub, nsmxsub, nsmysub;
  MPI_Comm comm;
  MPI_Request request[4];

  webdata = (UserData) rdata;
  cdata = N_VDATA(cc);

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

  comm = webdata->comm;     thispe = webdata->thispe;
  ixsub = webdata->ixsub;   jysub = webdata->jysub;
  cext = webdata->cext;
  nsmxsub = webdata->nsmxsub; nsmysub = (webdata->ns)*(webdata->mysub);

  /* Start receiving boundary data from neighboring PEs. */

  BRecvPost(comm, request, thispe, ixsub, jysub, nsmxsub, nsmysub,
            cext, buffer);

  /* Send data from boundary of local grid to neighboring PEs. */

  BSend(comm, thispe, ixsub, jysub, nsmxsub, nsmysub, cdata);

  /* Finish receiving boundary data from neighboring PEs. */

  BRecvWait(request, ixsub, jysub, nsmxsub, cext, buffer);

  return(0);

} /* End of rescomm. */


/*************************************************************************/
/* BSend: Send boundary data to neighboring PEs.
   This routine sends components of cc from internal subgrid boundaries
   to the appropriate neighbor PEs.                                      */
```

```
static void BSend(MPI_Comm comm, integer my_pe, integer ixsub, integer jysub,
                  integer dsizex, integer dsizey, real cdata[])
{
  int i;
  integer ly, offsetc, offsetbuf;
  real bufleft[NUM_SPECIES*MYSUB], bufright[NUM_SPECIES*MYSUB];

  /* If jysub > 0, send data from bottom x-line of cc. */

  if (jysub != 0)
    MPI_Send(&cdata[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);

  /* If jysub < NPEY-1, send data from top x-line of cc. */

  if (jysub != NPEY-1) {
    offsetc = (MYSUB-1)*dsizex;
    MPI_Send(&cdata[offsetc], dsizex, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
  }

  /* If ixsub > 0, send data from left y-line of cc (via bufleft). */

  if (ixsub != 0) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NUM_SPECIES;
      offsetc = ly*dsizex;
      for (i = 0; i < NUM_SPECIES; i++)
        bufleft[offsetbuf+i] = cdata[offsetc+i];
    }
    MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
  }

  /* If ixsub < NPEX-1, send data from right y-line of cc (via bufright). */

  if (ixsub != NPEX-1) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NUM_SPECIES;
      offsetc = offsetbuf*MXSUB + (MXSUB-1)*NUM_SPECIES;
      for (i = 0; i < NUM_SPECIES; i++)
        bufright[offsetbuf+i] = cdata[offsetc+i];
    }
    MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
  }

} /* End of Bsend. */
```

```
/*********************************************************************/
/* BRecvPost: Start receiving boundary data from neighboring PEs.
   (1) buffer should be able to hold 2*NUM_SPECIES*MYSUB real entries,
   should be passed to both the BRecvPost and BRecvWait functions, and
   should not be manipulated between the two calls.
   (2) request should have 4 entries, and is also passed in both calls.  */

static void BRecvPost(MPI_Comm comm, MPI_Request request[], integer my_pe,
      integer ixsub, integer jysub,
      integer dsizex, integer dsizey,
      real cext[], real buffer[])
{
  integer offsetce;
  /* Have bufleft and bufright use the same buffer. */
  real *bufleft = buffer, *bufright = buffer+NUM_SPECIES*MYSUB;

  /* If jysub > 0, receive data for bottom x-line of cext. */
  if (jysub != 0)
    MPI_Irecv(&cext[NUM_SPECIES], dsizex, PVEC_REAL_MPI_TYPE,
      my_pe-NPEX, 0, comm, &request[0]);

  /* If jysub < NPEY-1, receive data for top x-line of cext. */
  if (jysub != NPEY-1) {
    offsetce = NUM_SPECIES*(1 + (MYSUB+1)*(MXSUB+2));
    MPI_Irecv(&cext[offsetce], dsizex, PVEC_REAL_MPI_TYPE,
                                       my_pe+NPEX, 0, comm, &request[1]);
  }

  /* If ixsub > 0, receive data for left y-line of cext (via bufleft). */
  if (ixsub != 0) {
    MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
                                       my_pe-1, 0, comm, &request[2]);
  }

 /* If ixsub < NPEX-1, receive data for right y-line of cext (via bufright). */
  if (ixsub != NPEX-1) {
    MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
                                       my_pe+1, 0, comm, &request[3]);
  }

} /* End of BRecvPost. */


/*********************************************************************/
/* BRecvWait: Finish receiving boundary data from neighboring PEs.
   (1) buffer should be able to hold 2*NUM_SPECIES*MYSUB real entries,
   should be passed to both the BRecvPost and BRecvWait functions, and
   should not be manipulated between the two calls.
```

```
   (2) request should have 4 entries, and is also passed in both calls.  */

static void BRecvWait(MPI_Request request[], integer ixsub, integer jysub,
      integer dsizex, real cext[], real buffer[])
{
  int i;
  integer ly, dsizex2, offsetce, offsetbuf;
  real *bufleft = buffer, *bufright = buffer+NUM_SPECIES*MYSUB;
  MPI_Status status;

  dsizex2 = dsizex + 2*NUM_SPECIES;

  /* If jysub > 0, receive data for bottom x-line of cext. */
  if (jysub != 0)
    MPI_Wait(&request[0],&status);

  /* If jysub < NPEY-1, receive data for top x-line of cext. */
  if (jysub != NPEY-1)
    MPI_Wait(&request[1],&status);

  /* If ixsub > 0, receive data for left y-line of cext (via bufleft). */
  if (ixsub != 0) {
    MPI_Wait(&request[2],&status);

    /* Copy the buffer to cext */
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NUM_SPECIES;
      offsetce = (ly+1)*dsizex2;
      for (i = 0; i < NUM_SPECIES; i++)
        cext[offsetce+i] = bufleft[offsetbuf+i];
    }
  }

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

    /* Copy the buffer to cext */
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NUM_SPECIES;
      offsetce = (ly+2)*dsizex2 - NUM_SPECIES;
      for (i = 0; i < NUM_SPECIES; i++)
        cext[offsetce+i] = bufright[offsetbuf+i];
    }
  }
} /* End of BRecvWait. */
```

```
/* Define lines are for ease of readability in the following functions. */

#define mxsub      (webdata->mxsub)
#define mysub      (webdata->mysub)
#define npex       (webdata->npex)
#define npey       (webdata->npey)
#define ixsub      (webdata->ixsub)
#define jysub      (webdata->jysub)
#define nsmxsub    (webdata->nsmxsub)
#define nsmxsub2   (webdata->nsmxsub2)
#define np         (webdata->np)
#define dx         (webdata->dx)
#define dy         (webdata->dy)
#define cox        (webdata->cox)
#define coy        (webdata->coy)
#define rhs        (webdata->rhs)
#define cext       (webdata->cext)
#define rates      (webdata->rates)
#define ns         (webdata->ns)
#define acoef      (webdata->acoef)
#define bcoef      (webdata->bcoef)


/*********************************************************************/
/* reslocal: Compute res = F(t,cc,cp).
   This routine assumes that all inter-processor communication of data
   needed to calculate F has already been done.  Components at interior
   subgrid boundaries are assumed to be in the work array cext.
   The local portion of the cc vector is first copied into cext.
   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 reaction terms are saved separately in the vector webdata->rates
   for use by the preconditioner setup routine.                      */

static int reslocal(real tt, N_Vector cc, N_Vector cp, N_Vector res,
                    void *rdata)
{
  real *cdata, *ratesxy, *cpxy, *resxy,
       xx, yy, dcyli, dcyui, dcxli, dcxui;
  integer ix, jy, is, i, locc, ylocce, locce;
  UserData webdata;

  webdata = (UserData) rdata;

  /* Get data pointers, subgrid data, array sizes, work array cext. */
```

```
cdata = N_VDATA(cc);

/* Copy local segment of cc vector into the working extended array cext. */

locc = 0;
locce = nsmxsub2 + NUM_SPECIES;
for (jy = 0; jy < mysub; jy++) {
  for (i = 0; i < nsmxsub; i++) cext[locce+i] = cdata[locc+i];
  locc = locc + nsmxsub;
  locce = locce + nsmxsub2;
}

/* To facilitate homogeneous Neumann boundary conditions, when this is
a boundary PE, copy data from the first interior mesh line of cc to cext. */

/* If jysub = 0, copy x-line 2 of cc to cext. */
if (jysub == 0)
  { for (i = 0; i < nsmxsub; i++) cext[NUM_SPECIES+i] = cdata[nsmxsub+i]; }

/* If jysub = npey-1, copy x-line mysub-1 of cc to cext. */
if (jysub == npey-1) {
  locc = (mysub-2)*nsmxsub;
  locce = (mysub+1)*nsmxsub2 + NUM_SPECIES;
  for (i = 0; i < nsmxsub; i++) cext[locce+i] = cdata[locc+i];
}

/* If ixsub = 0, copy y-line 2 of cc to cext. */
if (ixsub == 0) {
  for (jy = 0; jy < mysub; jy++) {
    locc = jy*nsmxsub + NUM_SPECIES;
    locce = (jy+1)*nsmxsub2;
    for (i = 0; i < NUM_SPECIES; i++) cext[locce+i] = cdata[locc+i];
  }
}

/* If ixsub = npex-1, copy y-line mxsub-1 of cc to cext. */
if (ixsub == npex-1) {
  for (jy = 0; jy < mysub; jy++) {
    locc  = (jy+1)*nsmxsub - 2*NUM_SPECIES;
    locce = (jy+2)*nsmxsub2 - NUM_SPECIES;
    for (i = 0; i < NUM_SPECIES; i++) cext[locce+i] = cdata[locc+i];
  }
}

/* Loop over all grid points, setting local array rates to right-hand sides.
   Then set res values appropriately for prey/predator components of F. */

for (jy = 0; jy < mysub; jy++) {
```

```
    ylocce = (jy+1)*nsmxsub2;
    yy     = (jy+jysub*mysub)*dy;

    for (ix = 0; ix < mxsub; ix++) {
      locce = ylocce + (ix+1)*NUM_SPECIES;
      xx = (ix + ixsub*mxsub)*dx;

      ratesxy = IJ_Vptr(rates,ix,jy);
      WebRates(xx, yy, &(cext[locce]), ratesxy, webdata);

      resxy = IJ_Vptr(res,ix,jy);
      cpxy = IJ_Vptr(cp,ix,jy);

      for (is = 0; is < NUM_SPECIES; is++) {
        dcyli = cext[locce+is]          - cext[locce+is-nsmxsub2];
        dcyui = cext[locce+is+nsmxsub2] - cext[locce+is];

        dcxli = cext[locce+is]             - cext[locce+is-NUM_SPECIES];
        dcxui = cext[locce+is+NUM_SPECIES] - cext[locce+is];

        rhs[is] = cox[is]*(dcxui-dcxli) + coy[is]*(dcyui-dcyli) + ratesxy[is];

        if (is < np) resxy[is] = cpxy[is] - rhs[is];
        else         resxy[is] =          - rhs[is];

      } /* End of is (species) loop. */
    } /* End of ix loop. */
  } /* End of jy loop. */

  return(0);

} /* End of reslocal. */


/*****************************************************************/
/* WebRates: Evaluate reaction rates at a given spatial point.   */
/* At a given (x,y), evaluate the array of ns reaction terms R.  */

static void WebRates(real xx, real yy, real *cxy, real *ratesxy,
                     UserData webdata)
{
  int is;
  real fac;

  for (is = 0; is < NUM_SPECIES; is++)
       ratesxy[is] = dotprod(NUM_SPECIES, cxy, acoef[is]);

  fac = ONE + ALPHA*xx*yy + BETA*sin(FOURPI*xx)*sin(FOURPI*yy);
```

```
  for (is = 0; is < NUM_SPECIES; is++)
       ratesxy[is] = cxy[is]*( bcoef[is]*fac + ratesxy[is] );

} /* End of WebRates. */


/*****************************************************************/
/* dotprod: dot product routine for real arrays, for use by WebRates.    */

static real dotprod(integer size, real *x1, real *x2)
{
  integer i;
  real *xx1, *xx2, temp = ZERO;

  xx1 = x1; xx2 = x2;
  for (i = 0; i < size; i++) temp += (*xx1++) * (*xx2++);
  return(temp);

} /* End of dotprod. */


/*****************************************************************/
/* Preconbd: Preconditioner setup routine.
   This routine generates and preprocesses the block-diagonal
   preconditoner PP.  At each spatial point, a block of PP is computed
   by way of difference quotients on the reaction rates R.
   The base value of R are taken from webdata->rates, as set by webres.
   Each block is LU-factored, for later solution of the linear systems.  */

static int Precondbd(integer Neq, real tt, N_Vector cc, N_Vector cp,
                     N_Vector rr, real cj, ResFn res,  void *rdata,
                     void *Pdata, N_Vector ewt, N_Vector constraints,
                     real hh, real uround, long int *nrePtr,
                     N_Vector tempv1, N_Vector tempv2, N_Vector tempv3)

{
  real xx, yy, *cxy, *ewtxy, cctemp, **Pxy, *ratesxy, *Pxycol, *cpxy;
  real inc, sqru, fac, perturb_rates[NUM_SPECIES];
  integer is, js, ix, jy, ret;
  UserData webdata;

  webdata = (UserData)Pdata;
  sqru = RSqrt(uround);

  for (jy = 0; jy < mysub; jy++) {
    yy = (jy + jysub*mysub)*dy;
```

```
    for (ix = 0; ix < mxsub; ix++) {
      xx = (ix+ ixsub*mxsub)*dx;
      Pxy = (webdata->PP)[ix][jy];
      cxy = IJ_Vptr(cc,ix,jy);
      cpxy = IJ_Vptr(cp,ix,jy);
      ewtxy= IJ_Vptr(ewt,ix,jy);
      ratesxy = IJ_Vptr(rates,ix,jy);

      for (js = 0; js < ns; js++) {
        inc = sqru*(MAX(ABS(cxy[js]), MAX(hh*ABS(cpxy[js]), ONE/ewtxy[js])));
        cctemp = cxy[js];  /* Save the (js,ix,jy) element of cc. */
        cxy[js] += inc;    /* Perturb the (js,ix,jy) element of cc. */
        fac = -ONE/inc;

        WebRates(xx, yy, cxy, perturb_rates, webdata);

        Pxycol = Pxy[js];

        for (is = 0; is < ns; is++)
          Pxycol[is] = (perturb_rates[is] - ratesxy[is])*fac;

        if (js < np) Pxycol[js] += cj; /* Add partial with respect to cp. */

        cxy[js] = cctemp; /* Restore (js,ix,jy) element of cc. */

      } /* End of js loop. */

      /* Do LU decomposition of matrix block for grid point (ix,jy). */

      ret = gefa(Pxy, ns, (webdata->pivot)[ix][jy]);

      if (ret != 0) return(1);

    } /* End of ix loop. */
  } /* End of jy loop. */

  return(0);

} /* End of Precondbd. */


/*************************************************************************/
/* PSolvebd: Preconditioner solve routine.
   This routine applies the LU factorization of the blocks of the
   preconditioner PP, to compute the solution of PP * zvec = rvec.       */

static int PSolvebd(integer Neq, real tt, N_Vector cc, N_Vector cp,
                    N_Vector rr, real cj,ResFn res, void *rdata,
```

```
                    void *Pdata, N_Vector ewt, real delta,
                    N_Vector rvec, N_Vector zvec,
                    long int *nfePtr, N_Vector tempv)
{
  real **Pxy, *zxy;
  integer *pivot, ix, jy;
  UserData webdata;

  webdata = (UserData)Pdata;

  N_VScale(ONE, rvec, zvec);

  /* Loop through subgrid and apply preconditioner factors at each point. */
  for (ix = 0; ix < mxsub; ix++) {
    for (jy = 0; jy < mysub; jy++) {
      /* For grid point (ix,jy), do backsolve on local vector.
         zxy is the address of the local portion of zvec, and
         Pxy is the address of the corresponding block of PP.  */
      zxy = IJ_Vptr(zvec,ix,jy);
      Pxy = (webdata->PP)[ix][jy];
      pivot = (webdata->pivot)[ix][jy];
      gesl(Pxy, ns, pivot, zxy);

    } /* End of jy loop. */
  } /* End of ix loop. */

  return(0);

} /* End of PSolvebd. */
```

Sample output for the example program `webpk`

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

Number of species ns: 2     Mesh dimensions: 20 x 20     Total system size: 800
Subgrid dimensions: 10 x 10     Processor array: 2 x 2
Tolerance parameters:  rtol = 1e-05   atol = 1e-05
Linear solver: IDASPGMR     Max. Krylov dimension maxl: 10
Preconditioner: block diagonal, block size ns, via difference quotients


TIME t = 0.000000e+00.     NST = 0,  k = 0,  h = 1.631027e-08
NRE = 17,  NNI = 3,  NLI = 12,  NPE = 2,  NPS = 17
At bottom left:  c1, c2 = 1.000000e+01 9.999900e+04
At top right:    c1, c2 = 1.000000e+01 9.994900e+04
```

```
TIME t = 1.000000e-03.     NST = 33,  k = 4,  h = 6.012614e-05
NRE = 101,  NNI = 47,  NLI = 52,  NPE = 13,  NPS = 101
At bottom left:  c1, c2 = 1.031834e+01 1.031876e+05
At top right:    c1, c2 = 1.082687e+01 1.082227e+05


TIME t = 1.000000e-02.     NST = 103,  k = 5,  h = 2.164541e-04
NRE = 506,  NNI = 126,  NLI = 378,  NPE = 14,  NPS = 506
At bottom left:  c1, c2 = 1.618873e+02 1.618888e+06
At top right:    c1, c2 = 1.973486e+02 1.973449e+06


TIME t = 1.000000e-01.     NST = 169,  k = 1,  h = 2.218267e-02
NRE = 854,  NNI = 204,  NLI = 648,  NPE = 21,  NPS = 854
At bottom left:  c1, c2 = 2.401904e+02 2.401915e+06
At top right:    c1, c2 = 2.707209e+02 2.707169e+06


TIME t = 4.000000e-01.     NST = 172,  k = 1,  h = 1.774613e-01
NRE = 870,  NNI = 207,  NLI = 661,  NPE = 24,  NPS = 870
At bottom left:  c1, c2 = 2.401904e+02 2.401915e+06
At top right:    c1, c2 = 2.707209e+02 2.707169e+06


TIME t = 7.000000e-01.     NST = 173,  k = 1,  h = 3.549227e-01
NRE = 874,  NNI = 208,  NLI = 664,  NPE = 25,  NPS = 874
At bottom left:  c1, c2 = 2.401904e+02 2.401915e+06
At top right:    c1, c2 = 2.707209e+02 2.707169e+06


TIME t = 1.000000e+00.     NST = 174,  k = 1,  h = 7.098454e-01
NRE = 879,  NNI = 209,  NLI = 668,  NPE = 26,  NPS = 879
At bottom left:  c1, c2 = 2.401904e+02 2.401915e+06
At top right:    c1, c2 = 2.707209e+02 2.707169e+06


Final statistics:

NST  =   174     NRE  =   879
NNI  =   209     NLI  =   668
NPE  =    26     NPS  =   879
NETF =     1     NCFN =     0     NCFL =     0
```