Example Programs for KINSOL v2.2.1

Aaron M. Collier and Radu Serban
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

January 2005

UCRL-SM-208114

DISCLAIMER

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

Contents

1 Introduction

This report is intended to serve as a companion document to the User Documentation of KINSOL [1]. It provides details, with listings, on the example programs supplied with the KINSOL distribution package.

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

Supplied in the `sundials/kinsol/examples_ser` directory is the following serial example (using the NVECTOR_SERIAL module):

- `kinwebs` is an example program for KINSOL with the Krylov linear solver.

  This program solves a nonlinear system that arises from a system of partial differential equations. The PDE system is a six-species food web population model, with predator-prey interaction and diffusion on the unit square in two dimensions. The preconditioner is a block-diagonal matrix based on the partial derivatives of the interaction terms only.

Supplied in the `sundials/kinsol/examples_par` directory are the following two parallel examples (using the NVECTOR_PARALLEL module):

- `kinwebp` is a parallel implementation of `kinwebs`.
- `kinwebbbd` solves the same problem as `kinwebp`, with a block-diagonal matrix with banded blocks as a preconditioner, generated by difference quotients, using the module KINBBDPRE.

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

- `kindiagsf` is a serial example, which solves a nonlinear system of the form $u_i^2 = i^2$ using an approximate diagonal preconditioner.
- `kindiagpf` is a parallel implementation of `kindiagsf`.

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

In the descriptions below, we make frequent references to the KINSOL User Document [1]. All citations to specific sections (e.g. §5.1) are references to parts of that User Document, unless explicitly stated otherwise.

Note. The examples in the KINSOL distribution are written in such a way as to compile and run for any combination of configuration options during the installation of SUNDIALS (see §2). As a consequence, they contain portions of code that will not be typically

present in a user program. For example, all C example programs make use of the variable `SUNDIALS_EXTENDED_PRECISION` to test if the solver libraries were built in extended precision and use the appropriate conversion specifiers in `printf` functions. Similarly, the FORTRAN examples in FKINSOL are automatically pre-processed to generate source code that corresponds to the precision in which the KINSOL libraries were built (see §3 in this document for more details).

2 C example problems

2.1 A serial example: `kinwebs`

We give here an example that illustrates the use of KINSOL with the Krylov method SPGMR, in the KINSPGMR module, as the linear system solver. The source file, `kinwebs.c`, is listed in Appendix A.

This program solves a nonlinear system that arises from a discretized system of partial differential equations. The PDE system is a six-species food web population model, with predator-prey interaction and diffusion on the unit square in two dimensions. Given the dependent variable vector of species concentrations $c = [c_1, c_2, ..., c_{n_s}]^T$, where $n_s = 2n_p$ is the number of species and n_p is the number of predators and of prey, then the PDEs can be written as

$$d_i \cdot \left(\frac{\partial^2 c_i}{\partial x^2} + \frac{\partial^2 c_i}{\partial y^2} \right) + f_i(x, y, c) = 0 \quad (i = 1, ..., n_s)\,, \tag{1}$$

where the subscripts i are used to distinguish the species, and where

$$f_i(x, y, c) = c_i \cdot \left(b_i + \sum_{j=1}^{n_s} a_{i,j} \cdot c_j \right)\,. \tag{2}$$

The problem coefficients are given by

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

$$b_i = b_i(x, y) = \begin{cases} 1 + \alpha xy & i \le n_p \\ -1 - \alpha xy & i > n_p\,, \end{cases}$$

and

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

The spatial domain is the unit square $(x, y) \in [0, 1] \times [0, 1]$.

Homogeneous Neumann boundary conditions are imposed and the initial guess is constant in both x and y. For this example, the equations (1) are discretized spatially with standard central finite differences on a 8×8 mesh with $n_s = 6$, giving a system of size 384.

Among the initial `#include` lines in this case are lines to include `kinspgmr.h` and `sundialsmath.h`. The first contains constants and function prototypes associated with the SPGMR method. The inclusion of `sundialsmath.h` is done to access the `MAX` and `ABS` macros, and the `RSqrt` function to compute the square root of a `realtype` number.

The `main` program calls `KINCreate` and then calls `KINMalloc` with the name of the user-supplied system function `func` and solution vector as arguments. The `main` program then calls a number of `KINSet*` routines to notify KINSOL of the function data pointer, the positivity constraints on the solution, and convergence tolerances on the system function and step size. It calls `KINSpgmr` (see §5.4.2) to specify the KINSPGMR linear solver, and passes a value of 15 as the maximum Krylov subspace dimension, `maxl`. Next, a

maximum value of maxlrst = 2 restarts is imposed and the user-supplied preconditioner setup and solve functions, PrecSetupBD and PrecSolveBD, are specified through calls to KINSpgmrSetPrecSetupFn and KINSpgmrSetPrecSolveFn, respectively (see §5.4.4). The data pointer passed to KINSpgmrSetPrecData is passed to PrecSetupBD and PrecSolveBD whenever these are called.

Next, KINSol is called, the return value is tested for error conditions, and the approximate solution vector is printed via a call to PrintOutput. After that, PrintFinalStats is called to get and print final statistics, and memory is freed by calls to N_VDestroy_Serial, FreeUserData and KINFree. The statistics printed are the total numbers of nonlinear iterations (nni), of func evaluations (excluding those for Jv product evaluations) (nfe), of func evaluations for Jv evaluations (nfeSG), of linear (Krylov) iterations (nli), of preconditioner evaluations (npe), and of preconditioner solves (nps). All of these optional outputs and others are described in §5.4.5.

Mathematically, the dependent variable has three dimensions: species number, x mesh point, and y mesh point. But in NVECTOR_SERIAL, a vector of type N_Vector works with a one-dimensional contiguous array of data components. The macro IJ_Vptr isolates the translation from three dimensions to one. Its use results in clearer code and makes it easy to change the underlying layout of the three-dimensional data. Here the problem size is 384, so we use the NV_DATA_S macro for efficient N_Vector access. The NV_DATA_S macro gives a pointer to the first component of a serial N_Vector which is then passed to the IJ_Vptr macro.

The preconditioner used here is the block-diagonal part of the true Newton matrix and is based only on the partial derivatives of the interaction terms f in (2) and hence its diagonal blocks are $n_s \times n_s$ matrices ($n_s = 6$). It is generated and factored in the PrecSetupBD routine and backsolved in the PrecSolveBD routine. See §5.5.4 for detailed descriptions of these preconditioner functions.

The program kinwebs.c uses the "small" dense functions for all operations on the 6×6 preconditioner blocks. Thus it includes smalldense.h, and calls the small dense matrix functions denalloc, denallocpiv, denfree, denfreepiv, gefa, and gesl. The small dense functions are generally available for KINSOL user programs (for more information, see §8.1 or the comments in the header file smalldense.h).

In addition to the functions called by KINSOL, kinwebs.c includes definitions of several private functions. These are: AllocUserData to allocate space for P and the pivot arrays; InitUserData to load problem constants in the data block; FreeUserData to free that block; SetInitialProfiles to load the initial values in cc; PrintOutput to retreive and print selected solution values; PrintFinalStats to print statistics; and check_flag to check return values for error conditions.

The output generated by kinwebs is shown below. Note that the solution involved 7 Newton iterations, with an average of about 33 Krylov iterations per Newton iteration.

kinwebs sample output

```
Predator-prey test problem --  KINSol (serial version)

Mesh dimensions = 8 X 8
Number of species = 6
Total system size = 384

Flag globalstrategy = 1 (1 = Inex. Newton, 2 = Linesearch)
```

```
Linear solver is SPGMR with maxl = 15, maxlrst = 2
Preconditioning uses interaction-only block-diagonal matrix
Positivity constraints imposed on all components
Tolerance parameters:  fnormtol = 1e-07   scsteptol = 1e-13

Initial profile of concentration
At all mesh points:  1 1 1   30000 30000 30000



Computed equilibrium species concentrations:

At bottom left:
 1.16428 1.16428 1.16428 34927.5 34927.5 34927.5

At top right:
 1.25797 1.25797 1.25797 37736.7 37736.7 37736.7


Final Statistics..

nni    =     7    nli   =   230
nfe    =     8    nfeSG =   237
nps    =   237    npe   =     1     ncfl  =     4
```

2.2 A parallel example: `kinwebbbd`

In this example, `kinwebbbd`, we solve the same problem as with `kinwebs` above, but in parallel, and instead of supplying the preconditioner we use the KINBBDPRE module. The source is given in Appendix B.

KINBBDPRE generates and uses a band-block-diagonal preconditioner, generated by difference quotients. The upper and lower half-bandwidths of the Jacobian block on each process are both equal to 2·`NUM_SPECIES`−1, and that is the value supplied as `mu` and `ml` in the call to `KINBBDPrecAlloc`.

In this case, we think of the parallel MPI processes as being laid out in a rectangle, and each process being assigned a subgrid of size `MXSUB`×`MYSUB` of the $x - y$ grid. If there are `NPEX` processes in the x direction and `NPEY` processes in the y direction, then the overall grid size is `MX`×`MY` with `MX`=`NPEX`×`MXSUB` and `MY`=`NPEY`×`MYSUB`, and the size of the nonlinear system is `NUM_SPECIES`·`MX`·`MY`.

The evaluation of the nonlinear system function is performed in `func`. In this parallel setting, the processes first communicate the subgrid boundary data and then compute the local components of the nonlinear system function. The MPI communication is isolated in the private function `ccomm` (which in turn calls `BRecvPost`, `BSend`, and `BRecvWait`) and the subgrid boundary data received from neighboring processes is loaded into the work array `cext`. The computation of the nonlinear system function is done in `func_local` which starts by copying the local segment of the `cc` vector into `cext` and then by imposing the boundary conditions by copying the first interior mesh line from `cc` into `cext`. After this, the nonlinear system function is evaluated by using central finite-difference approximations using the data in `cext` exclusively.

The function `func_local` is also passed as the `gloc` argument to `KINBBDPrecAlloc`.

Since all communication needed for the evaluation of the local aproximation of f used in building the band-block-diagonal preconditioner is already done for the evaluation of f in `func`, a `NULL` pointer is passed as the `gcomm` argument to `KINBBDPrecAlloc`.

The `main` program resembles closely that of the `kinwebs` example, with particularization arising from the use of the parallel MPI NVECTOR_PARALLEL module. It begins by initializing MPI and obtaining the total number of processes and the id of the local process. The local length of the solution vector is then computed as `NUM_SPECIES`·`MXSUB`·`MYSUB`. Distributed vectors are created by calling the constructor defined in NVECTOR_PARALLEL with the MPI communicator and the local and global problem sizes as arguments. All output is performed only from the process with id equal to 0. Finally, after all memory deallocation, the MPI environment is terminated by calling `MPI_Finalize`.

The output generated by `kinwebbbd` is shown below.

kinwebbbd sample output

```
Predator-prey test problem--  KINSol (parallel-BBD version)

Mesh dimensions = 20 X 20
Number of species = 6
Total system size = 2400

Subgrid dimensions = 10 X 10
Processor array is 2 X 2

Flag globalstrategy = 1 (1 = Inex. Newton, 2 = Linesearch)
Linear solver is SPGMR with maxl = 20, maxlrst = 2
Preconditioning uses band-block-diagonal matrix from KINBBDPRE
  with matrix half-bandwidths ml, mu = 11  11
Tolerance parameters:  fnormtol = 1e-07   scsteptol = 1e-13

Initial profile of concentration
At all mesh points:  1 1 1   30000 30000 30000



Computed equilibrium species concentrations:

At bottom left:
 1.165 1.165 1.165 34949 34949 34949

At top right:
 1.25552 1.25552 1.25552 37663.2 37663.2 37663.2


Final Statistics..

nni    =    10    nli   =   540
nfe    =    11    nfeSG =   550
nps    =   550    npe   =     1     ncfl  =     7
```

3 Fortran example problems

The FORTRAN example problem programs supplied with the KINSOL package are all written in standard F77 Fortran and use double-precision arithmetic. However, when the FORTRAN examples are built, the source code is automatically modified according to the configure options supplied by the user and the system type. Integer variables are declared as `INTEGER*`n, where n denotes the number of bytes in the corresponding C type (`long int` or `int`). Floating-point variable declarations remain unchanged if double-precision is used, but are changed to `REAL*`n, where n denotes the number of bytes in the SUNDIALS type `realtype`, if using single-precision. Also, if using single-precision, declarations of floating-point constants are appropriately modified, e.g.; `0.5D-4` is changed to `0.5E-4`.

The two examples supplied with the FKINSOL module are very simple tests of the FORTRAN-C interface module. They solve the nonlinear system

$$F(u) = 0\,, \quad \text{where } f_i(u) = u_i^2 - i^2\,, 1 \le i \le N\,.$$

3.1 A serial example: `kindiagsf`

The `kindiagsf` program, for which the source code is listed in Appendix C, solves the above problem using the serial NVECTOR_SERIAL module.

The main program begins by calling `fnvinits` to initialize computations with the serial NVECTOR_SERIAL module. Next, the array `uu` is set to contain the initial guess $u_i = 2i$, the array `scale` is set with all components equal to 1.0 (meaning that no scaling is done), and the array `constr` is set with all components equal to 0.0 to indicate that no inequality constraints should be imposed on the solution vector.

The KINSOL solver is initialized and memory for it is allocated by calling `fkinmalloc`, which also specifies the maximum number of iterations between calls to the preconditioner setup routine (`msbpre` $= 5$), the tolerance for stopping based on the function norm (`fnormtol` $= 10^{-5}$), the tolerance for stopping based on the step length (`scsteptol` $= 10^{-4}$), and that no optional inputs are provided (`inopt` $= 0$).

Next, the KINSPGMR linear solver module is attached to KINSOL by calling `fkinspgmr`, which also specifies the maximum Krylov subspace dimension (`maxl` $= 10$) and the maximum number of restarts allowed for SPGMR (`maxlrst` $= 2$). The KINSPGMR module is directed to use the supplied preconditioner by calling the routines `fkinspgmrsetpsol` and `fkinspgmrsetpset` with a first argument equal to 1. The solution of the nonlinear system is obtained after a successful return from `fkinsol`, which is then printed to unit 6 (stdout).

Memory allocated for the KINSOL solver is released by calling `fkinfree` and computations with the NVECTOR_SERIAL module are terminated by calling `fnvfrees`.

The user-supplied routine `fkfun` contains a straightforward transcription of the nonlinear system function f, while the routine `fkpset` sets the array `pp` (in the common block `pcom`) to contain an approximation to the reciprocals of the Jacobian diagonal elements. The components of `pp` are then used in `fkpsol` to solve the preconditioner linear system $Px = v$ through simple multiplications.

The following is sample output from `kindiagsf`, using $N = 128$.

kindiagsf sample output

```
Example program kindiagsf:

 This fkinsol example code solves a 128 eqn diagonal algebraic system.
```

```
 Its purpose is to demonstrate the use of the Fortran interface
 in a serial environment.


 globalstrategy = KIN_INEXACT_NEWTON

 FKINSOL return code is   0


 The resultant values of uu are:

   1   1.000000   2.000000   3.000000   4.000000
   5   5.000000   6.000000   7.000000   8.000000
   9   9.000000  10.000000  11.000000  12.000000
  13  13.000000  14.000000  15.000000  16.000000
  17  17.000000  18.000000  19.000000  20.000000
  21  21.000000  22.000000  23.000000  24.000000
  25  25.000000  26.000000  27.000000  28.000000
  29  29.000000  30.000000  31.000000  32.000000
  33  33.000000  34.000000  35.000000  36.000000
  37  37.000000  38.000000  39.000000  40.000000
  41  41.000000  42.000000  43.000000  44.000000
  45  45.000000  46.000000  47.000000  48.000000
  49  49.000000  50.000000  51.000000  52.000000
  53  53.000000  54.000000  55.000000  56.000000
  57  57.000000  58.000000  59.000000  60.000000
  61  61.000000  62.000000  63.000000  64.000000
  65  65.000000  66.000000  67.000000  68.000000
  69  69.000000  70.000000  71.000000  72.000000
  73  73.000000  74.000000  75.000000  76.000000
  77  77.000000  78.000000  79.000000  80.000000
  81  81.000000  82.000000  83.000000  84.000000
  85  85.000000  86.000000  87.000000  88.000000
  89  89.000000  90.000000  91.000000  92.000000
  93  93.000000  94.000000  95.000000  96.000000
  97  97.000000  98.000000  99.000000 100.000000
 101 101.000000 102.000000 103.000000 104.000000
 105 105.000000 106.000000 107.000000 108.000000
 109 109.000000 110.000000 111.000000 112.000000
 113 113.000000 114.000000 115.000000 116.000000
 117 117.000000 118.000000 119.000000 120.000000
 121 121.000000 122.000000 123.000000 124.000000
 125 125.000000 126.000000 127.000000 128.000000


Final statistics:

 nni =    7,  nli =   21,  nfe =    8,  npe =    2,  nps =   28,  ncfl =    0
```

3.2 A parallel example: `kindiagpf`

The program `kindiapf`, listed in Appendix D, is a straightforward modification of `kindiagsf` to use the parallel MPI NVECTOR_PARALLEL module.

After initialization of MPI, the NVECTOR_PARALLEL module is initialized by calling `fnvinitp` with the local and global vector sizes as its first two arguments. The problem set-up (KINSOL initialization, KINSPGMR specification) and solution steps are the same as in `kindiagsf`. Upon successful return from `fkinsol`, the solution segment local to the process with id equal to 0 is printed to the screen. Finally, the KINSOL memory is released, NVECTOR_PARALLEL computations are finalized, and the MPI environent is terminated.

For this simple example, no inter-process communication is required to evaluate the nonlinear system function f or the preconditioner. As a consequence, the user-supplied routines `fkfun`, `fkpset`, and `fkpsol` are basically identical to those in `kindiagsf`.

Sample output from `kindiapf`, for $N = 128$, follows.

kindiagpf sample output

```
Example program kindiagpf:

 This fkinsol example code solves a 128 eqn diagonal algebraic system.
 Its purpose is to demonstrate the use of the Fortran interface
 in a parallel environment.


 globalstrategy = KIN_INEXACT_NEWTON

 FKINSOL return code is    0


 The resultant values of uu (process 0) are:

   1   1.000000   2.000000   3.000000   4.000000
   5   5.000000   6.000000   7.000000   8.000000
   9   9.000000  10.000000  11.000000  12.000000
  13  13.000000  14.000000  15.000000  16.000000
  17  17.000000  18.000000  19.000000  20.000000
  21  21.000000  22.000000  23.000000  24.000000
  25  25.000000  26.000000  27.000000  28.000000
  29  29.000000  30.000000  31.000000  32.000000


Final statistics:

 nni =    7,  nli =   21,  nfe =    8,  npe =    2,  nps=  28,  ncfl=   0
```

References

[1] A. M. Collier, A. C. Hindmarsh, R. Serban, and C.S. Woodward. User Documentation for KINSOL v2.2.0. Technical Report UCRL-SM-208116, LLNL, 2004.

A Listing of `kinwebs.c`

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.14.2.2 $
 * $Date: 2005/03/17 22:51:08 $
 * -----------------------------------------------------------------
 * Programmer(s): Allan Taylor, Alan Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example (serial):
 *
 * This example solves a nonlinear system that arises from a system
 * of partial differential equations. 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 the following:
 *
 *       1   2         ns
 * c = (c , c ,  ..., c  )     (denoted by the variable cc)
 *
 * and the PDE's are as follows:
 *
 *                     i       i
 *        0 = d(i)*(c     + c    )  +  f  (x,y,c)   (i=1,...,ns)
 *                     xx      yy         i
 *
 *   where
 *
 *                    i             ns         j
 *   f  (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 number np is both the
 * number of prey and predator species. 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)
 *   b(i) = BB * (1 + alpha * x * y)   (i <= np)
 *   b(i) =-BB * (1 + alpha * x * y)   (i >  np)
 *   d(i) = DPREY   (i <= np)
 *   d(i) = DPRED   ( i > np)
 *
 * The various scalar parameters are set using define's or in
 * routine InitUserData.
 *
 * The boundary conditions are: normal derivative = 0, and the
 * initial guess is constant in x and y, but the final solution
 * is not.
 *
 * The PDEs are discretized by central differencing on an MX by
```

```
 * MY mesh.
 *
 * The nonlinear system is solved by KINSOL using the method
 * specified in local variable globalstrat.
 *
 * The preconditioner matrix is a block-diagonal matrix based on
 * the partial derivatives of the interaction terms f only.
 *
 * Constraints are imposed to make all components of the solution
 * positive.
 * -----------------------------------------------------------------
 * References:
 *
 * 1. Peter N. Brown and Youcef Saad,
 *    Hybrid Krylov Methods for Nonlinear Systems of Equations
 *    LLNL report UCRL-97645, November 1987.
 *
 * 2. Peter N. Brown and Alan C. Hindmarsh,
 *    Reduced Storage Matrix Methods in Stiff ODE systems,
 *    Lawrence Livermore National Laboratory Report  UCRL-95088,
 *    Rev. 1, June 1987, and  Journal of Applied Mathematics and
 *    Computation, Vol. 31 (May 1989), pp. 40-91. (Presents a
 *    description of the time-dependent version of this test
 *    problem.)
 * -----------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "kinsol.h"         /* main KINSOL header file                          */
#include "kinspgmr.h"       /* use KINSPGMR linear solver                       */
#include "sundialstypes.h"  /* def's of realtype and booleantype                */
#include "nvector_serial.h" /* definitions of type N_Vector and access macros */
#include "iterative.h"      /* contains the enum for types of preconditioning */
#include "smalldense.h"     /* use generic DENSE solver for preconditioning   */
#include "sundialsmath.h"   /* contains RSqrt routine                           */

/* Problem Constants */

#define NUM_SPECIES     6  /* must equal 2*(number of prey or predators)
                              number of prey = number of predators       */

#define PI       RCONST(3.1415926535898)   /* pi */

#define MX          8              /* MX = number of x mesh points */
#define MY          8              /* MY = number of y mesh points */
#define NSMX        (NUM_SPECIES * MX)
#define NEQ         (NSMX * MY)    /* number of equations in the system */
#define AA          RCONST(1.0)    /* value of coefficient AA in above eqns */
#define EE          RCONST(10000.) /* value of coefficient EE in above eqns */
#define GG          RCONST(0.5e-6) /* value of coefficient GG in above eqns */
#define BB          RCONST(1.0)    /* value of coefficient BB in above eqns */
#define DPREY       RCONST(1.0)    /* value of coefficient dprey above */
```

```
#define DPRED       RCONST(0.5)    /* value of coefficient dpred above */
#define ALPHA       RCONST(1.0)    /* value of coefficient alpha above */
#define AX          RCONST(1.0)    /* total range of x variable */
#define AY          RCONST(1.0)    /* total range of y variable */
#define FTOL        RCONST(1.e-7)  /* ftol tolerance */
#define STOL        RCONST(1.e-13) /* stol tolerance */
#define THOUSAND    RCONST(1000.0) /* one thousand */
#define ZERO        RCONST(0.)     /* 0. */
#define ONE         RCONST(1.0)    /* 1. */
#define TWO         RCONST(2.0)    /* 2. */
#define PREYIN      RCONST(1.0)    /* initial guess for prey concentrations. */
#define PREDIN      RCONST(30000.0)/* initial guess for predator concs.      */

/* User-defined vector access macro: IJ_Vptr */

/* IJ_Vptr is defined in order to translate from the underlying 3D structure
   of the dependent variable vector to the 1D storage scheme for an N-vector.
   IJ_Vptr(vv,i,j) returns a pointer to the location in vv corresponding to
   indices is = 0, jx = i, jy = j.    */

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

/* Type : UserData
   contains preconditioner blocks, pivot arrays, and problem constants */

typedef struct {
  realtype **P[MX][MY];
  long int *pivot[MX][MY];
  realtype **acoef, *bcoef;
  N_Vector rates;
  realtype *cox, *coy;
  realtype ax, ay, dx, dy;
  realtype uround, sqruround;
  long int mx, my, ns, np;
} *UserData;

/* Functions Called by the KINSOL Solver */

static void func(N_Vector cc, N_Vector fval, void *f_data);

static int PrecSetupBD(N_Vector cc, N_Vector cscale,
                       N_Vector fval, N_Vector fscale,
                       void *P_data,
                       N_Vector vtemp1, N_Vector vtemp2);

static int PrecSolveBD(N_Vector cc, N_Vector cscale,
                       N_Vector fval, N_Vector fscale,
                       N_Vector vv, void *P_data,
                       N_Vector ftem);

/* Private Helper Functions */

static UserData AllocUserData(void);
static void InitUserData(UserData data);
```

```
161 static void FreeUserData(UserData data);
162 static void SetInitialProfiles(N_Vector cc, N_Vector sc);
163 static void PrintHeader(int globalstrategy, int maxl, int maxlrst,
164                         realtype fnormtol, realtype scsteptol);
165 static void PrintOutput(N_Vector cc);
166 static void PrintFinalStats(void *kmem);
167 static void WebRate(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
168                     void *f_data);
169 static realtype DotProd(long int size, realtype *x1, realtype *x2);
170 static int check_flag(void *flagvalue, char *funcname, int opt);
171
172 /*
173  *--------------------------------------------------------------------
174  * MAIN PROGRAM
175  *--------------------------------------------------------------------
176  */
177
178 int main()
179 {
180   int globalstrategy;
181   realtype fnormtol, scsteptol;
182   N_Vector cc, sc, constraints;
183   UserData data;
184   int flag, maxl, maxlrst;
185   void *kmem;
186
187   cc = sc = constraints = NULL;
188   kmem = NULL;
189   data = NULL;
190
191   /* Allocate memory, and set problem data, initial values, tolerances */
192   globalstrategy = KIN_INEXACT_NEWTON;
193
194   data = AllocUserData();
195   if (check_flag((void *)data, "AllocUserData", 2)) return(1);
196   InitUserData(data);
197
198   /* Create serial vectors of length NEQ */
199   cc = N_VNew_Serial(NEQ);
200   if (check_flag((void *)cc, "N_VNew_Serial", 0)) return(1);
201   sc = N_VNew_Serial(NEQ);
202   if (check_flag((void *)sc, "N_VNew_Serial", 0)) return(1);
203   data->rates = N_VNew_Serial(NEQ);
204   if (check_flag((void *)data->rates, "N_VNew_Serial", 0)) return(1);
205
206   constraints = N_VNew_Serial(NEQ);
207   if (check_flag((void *)constraints, "N_VNew_Serial", 0)) return(1);
208   N_VConst(TWO, constraints);
209
210   SetInitialProfiles(cc, sc);
211
212   fnormtol=FTOL; scsteptol=STOL;
213
214   /* Call KINCreate/KINMalloc to initialize KINSOL:
```

```
215       nvSpec is the nvSpec pointer used in the serial version
216       A pointer to KINSOL problem memory is returned and stored in kmem. */
217    kmem = KINCreate();
218    if (check_flag((void *)kmem, "KINCreate", 0)) return(1);
219    /* Vector cc passed as template vector. */
220    flag = KINMalloc(kmem, func, cc);
221    if (check_flag(&flag, "KINMalloc", 1)) return(1);
222
223    flag = KINSetFdata(kmem, data);
224    if (check_flag(&flag, "KINSetFdata", 1)) return(1);
225    flag = KINSetConstraints(kmem, constraints);
226    if (check_flag(&flag, "KINSetConstraints", 1)) return(1);
227    flag = KINSetFuncNormTol(kmem, fnormtol);
228    if (check_flag(&flag, "KINSetFuncNormTol", 1)) return(1);
229    flag = KINSetScaledStepTol(kmem, scsteptol);
230    if (check_flag(&flag, "KINSetScaledStepTol", 1)) return(1);
231
232    /* Call KINSpgmr to specify the linear solver KINSPGMR with preconditioner
233       routines PrecSetupBD and PrecSolveBD, and the pointer to the user block data. */
234    maxl = 15;
235    maxlrst = 2;
236    flag = KINSpgmr(kmem, maxl);
237    if (check_flag(&flag, "KINSpgmr", 1)) return(1);
238
239    flag = KINSpgmrSetMaxRestarts(kmem, maxlrst);
240    if (check_flag(&flag, "KINSpgmrSetMaxRestarts", 1)) return(1);
241    flag = KINSpgmrSetPrecSetupFn(kmem, PrecSetupBD);
242    if (check_flag(&flag, "KINSpgmrSetPrecSetupFn", 1)) return(1);
243    flag = KINSpgmrSetPrecSolveFn(kmem, PrecSolveBD);
244    if (check_flag(&flag, "KINSpgmrSetPrecSolveFn", 1)) return(1);
245    flag = KINSpgmrSetPrecData(kmem, data);
246    if (check_flag(&flag, "KINSpgmrSetPrecData", 1)) return(1);
247
248    /* Print out the problem size, solution parameters, initial guess. */
249    PrintHeader(globalstrategy, maxl, maxlrst, fnormtol, scsteptol);
250
251    /* Call KINSol and print output concentration profile */
252    flag = KINSol(kmem,           /* KINSol memory block */
253                  cc,             /* initial guess on input; solution vector */
254                  globalstrategy, /* global stragegy choice */
255                  sc,             /* scaling vector, for the variable cc */
256                  sc);            /* scaling vector for function values fval */
257    if (check_flag(&flag, "KINSol", 1)) return(1);
258
259    printf("\n\n\nComputed equilibrium species concentrations:\n");
260    PrintOutput(cc);
261
262    /* Print final statistics and free memory */
263    PrintFinalStats(kmem);
264
265    N_VDestroy_Serial(cc);
266    N_VDestroy_Serial(sc);
267    N_VDestroy_Serial(constraints);
268    KINFree(kmem);
```

```
  FreeUserData(data);

  return(0);
}

/* Readability definitions used in other routines below */

#define acoef  (data->acoef)
#define bcoef  (data->bcoef)
#define cox    (data->cox)
#define coy    (data->coy)

/*
 *--------------------------------------------------------------------
 * FUNCTIONS CALLED BY KINSOL
 *--------------------------------------------------------------------
 */

/*
 * System function for predator-prey system
 */

static void func(N_Vector cc, N_Vector fval, void *f_data)
{
  realtype xx, yy, delx, dely, *cxy, *rxy, *fxy, dcyli, dcyui, dcxli, dcxri;
  long int jx, jy, is, idyu, idyl, idxr, idxl;
  UserData data;

  data = (UserData)f_data;
  delx = data->dx;
  dely = data->dy;

  /* Loop over all mesh points, evaluating rate array at each point*/
  for (jy = 0; jy < MY; jy++) {

    yy = dely*jy;

    /* Set lower/upper index shifts, special at boundaries. */
    idyl = (jy != 0   ) ? NSMX : -NSMX;
    idyu = (jy != MY-1) ? NSMX : -NSMX;

    for (jx = 0; jx < MX; jx++) {

      xx = delx*jx;

      /* Set left/right index shifts, special at boundaries. */
      idxl = (jx !=  0  ) ?  NUM_SPECIES : -NUM_SPECIES;
      idxr = (jx != MX-1) ?  NUM_SPECIES : -NUM_SPECIES;

      cxy = IJ_Vptr(cc,jx,jy);
      rxy = IJ_Vptr(data->rates,jx,jy);
      fxy = IJ_Vptr(fval,jx,jy);

      /* Get species interaction rate array at (xx,yy) */
```

```
323       WebRate(xx, yy, cxy, rxy, f_data);
324
325       for(is = 0; is < NUM_SPECIES; is++) {
326
327         /* Differencing in x direction */
328         dcyli = *(cxy+is) - *(cxy - idyl + is) ;
329         dcyui = *(cxy + idyu + is) - *(cxy+is);
330
331         /* Differencing in y direction */
332         dcxli = *(cxy+is) - *(cxy - idxl + is);
333         dcxri = *(cxy + idxr +is) - *(cxy+is);
334
335         /* Compute the total rate value at (xx,yy) */
336         fxy[is] = (coy)[is] * (dcyui - dcyli) +
337           (cox)[is] * (dcxri - dcxli) + rxy[is];
338
339       } /* end of is loop */
340
341     } /* end of jx loop */
342
343   } /* end of jy loop */
344 }
345
346 /*
347  * Preconditioner setup routine. Generate and preprocess P.
348  */
349
350 static int PrecSetupBD(N_Vector cc, N_Vector cscale,
351                        N_Vector fval, N_Vector fscale,
352                        void *P_data,
353                        N_Vector vtemp1, N_Vector vtemp2)
354 {
355   realtype r, r0, uround, sqruround, xx, yy, delx, dely, csave, fac;
356   realtype *cxy, *scxy, **Pxy, *ratesxy, *Pxycol, perturb_rates[NUM_SPECIES];
357   long int i, j, jx, jy, ret;
358   UserData data;
359
360   data = (UserData) P_data;
361   delx = data->dx;
362   dely = data->dy;
363
364   uround = data->uround;
365   sqruround = data->sqruround;
366   fac = N_VWL2Norm(fval, fscale);
367   r0 = THOUSAND * uround * fac * NEQ;
368   if(r0 == ZERO) r0 = ONE;
369
370   /* Loop over spatial points; get size NUM_SPECIES Jacobian block at each */
371   for (jy = 0; jy < MY; jy++) {
372     yy = jy*dely;
373
374     for (jx = 0; jx < MX; jx++) {
375       xx = jx*delx;
376       Pxy = (data->P)[jx][jy];
```

```
377         cxy = IJ_Vptr(cc,jx,jy);
378         scxy= IJ_Vptr(cscale,jx,jy);
379         ratesxy = IJ_Vptr((data->rates),jx,jy);
380 
381         /* Compute difference quotients of interaction rate fn. */
382         for (j = 0; j < NUM_SPECIES; j++) {
383 
384           csave = cxy[j];  /* Save the j,jx,jy element of cc */
385           r = MAX(sqruround*ABS(csave), r0/scxy[j]);
386           cxy[j] += r; /* Perturb the j,jx,jy element of cc */
387           fac = ONE/r;
388 
389           WebRate(xx, yy, cxy, perturb_rates, data);
390 
391           /* Restore j,jx,jy element of cc */
392           cxy[j] = csave;
393 
394           /* Load the j-th column of difference quotients */
395           Pxycol = Pxy[j];
396           for (i = 0; i < NUM_SPECIES; i++)
397             Pxycol[i] = (perturb_rates[i] - ratesxy[i]) * fac;
398 
399 
400         } /* end of j loop */
401 
402         /* Do LU decomposition of size NUM_SPECIES preconditioner block */
403         ret = gefa(Pxy, NUM_SPECIES, (data->pivot)[jx][jy]);
404         if (ret != 0) return(1);
405 
406       } /* end of jx loop */
407 
408     } /* end of jy loop */
409 
410     return(0);
411   }
412 
413   /*
414    * Preconditioner solve routine
415    */
416 
417   static int PrecSolveBD(N_Vector cc, N_Vector cscale,
418                          N_Vector fval, N_Vector fscale,
419                          N_Vector vv, void *P_data,
420                          N_Vector ftem)
421   {
422     realtype **Pxy, *vxy;
423     long int *piv, jx, jy;
424     UserData data;
425 
426     data = (UserData)P_data;
427 
428     for (jx=0; jx<MX; jx++) {
429 
430       for (jy=0; jy<MY; jy++) {
```

```
431
432       /* For each (jx,jy), solve a linear system of size NUM_SPECIES.
433          vxy is the address of the corresponding portion of the vector vv;
434          Pxy is the address of the corresponding block of the matrix P;
435          piv is the address of the corresponding block of the array pivot. */
436       vxy = IJ_Vptr(vv,jx,jy);
437       Pxy = (data->P)[jx][jy];
438       piv = (data->pivot)[jx][jy];
439       gesl (Pxy, NUM_SPECIES, piv, vxy);
440
441     } /* end of jy loop */
442
443   } /* end of jx loop */
444
445   return(0);
446 }
447
448 /*
449  * Interaction rate function routine
450  */
451
452 static void WebRate(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
453                     void *f_data)
454 {
455   long int i;
456   realtype fac;
457   UserData data;
458
459   data = (UserData)f_data;
460
461   for (i = 0; i<NUM_SPECIES; i++)
462     ratesxy[i] = DotProd(NUM_SPECIES, cxy, acoef[i]);
463
464   fac = ONE + ALPHA * xx * yy;
465
466   for (i = 0; i < NUM_SPECIES; i++)
467     ratesxy[i] = cxy[i] * ( bcoef[i] * fac + ratesxy[i] );
468 }
469
470 /*
471  * Dot product routine for realtype arrays
472  */
473
474 static realtype DotProd(long int size, realtype *x1, realtype *x2)
475 {
476   long int i;
477   realtype *xx1, *xx2, temp = ZERO;
478
479   xx1 = x1; xx2 = x2;
480   for (i = 0; i < size; i++) temp += (*xx1++) * (*xx2++);
481
482   return(temp);
483 }
484
```

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

/*
 * Allocate memory for data structure of type UserData
 */

static UserData AllocUserData(void)
{
  int jx, jy;
  UserData data;

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

  for (jx=0; jx < MX; jx++) {
    for (jy=0; jy < MY; jy++) {
      (data->P)[jx][jy] = denalloc(NUM_SPECIES);
      (data->pivot)[jx][jy] = denallocpiv(NUM_SPECIES);
    }
  }
  acoef = denalloc(NUM_SPECIES);
  bcoef = (realtype *)malloc(NUM_SPECIES * sizeof(realtype));
  cox   = (realtype *)malloc(NUM_SPECIES * sizeof(realtype));
  coy   = (realtype *)malloc(NUM_SPECIES * sizeof(realtype));

  return(data);
}

/*
 * Load problem constants in data
 */

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

  data->mx = MX;
  data->my = MY;
  data->ns = NUM_SPECIES;
  data->np = NUM_SPECIES/2;
  data->ax = AX;
  data->ay = AY;
  data->dx = (data->ax)/(MX-1);
  data->dy = (data->ay)/(MY-1);
  data->uround = UNIT_ROUNDOFF;
  data->sqruround = RSqrt(data->uround);

  /* Set up the coefficients a and b plus others found in the equations */
  np = data->np;

```

```
  dx2=(data->dx)*(data->dx); dy2=(data->dy)*(data->dy);

  for (i = 0; i < np; i++) {
    a1= &(acoef[i][np]);
    a2= &(acoef[i+np][0]);
    a3= &(acoef[i][0]);
    a4= &(acoef[i+np][np]);

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

    /* and then change the diagonal elements of acoef to -AA */
    acoef[i][i]=-AA;
    acoef[i+np][i+np] = -AA;

    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;
  }
}

/*
 * Free data memory
 */

static void FreeUserData(UserData data)
{
  int jx, jy;

  for (jx=0; jx < MX; jx++) {
    for (jy=0; jy < MY; jy++) {
      denfree((data->P)[jx][jy]);
      denfreepiv((data->pivot)[jx][jy]);
    }
  }

  denfree(acoef);
  free(bcoef);
  free(cox);
  free(coy);
  N_VDestroy_Serial(data->rates);
  free(data);
}

```

```
593 /*
594  * Set initial conditions in cc
595  */
596
597 static void SetInitialProfiles(N_Vector cc, N_Vector sc)
598 {
599   int i, jx, jy;
600   realtype *cloc, *sloc;
601   realtype  ctemp[NUM_SPECIES], stemp[NUM_SPECIES];
602
603   /* Initialize arrays ctemp and stemp used in the loading process */
604   for (i = 0; i < NUM_SPECIES/2; i++) {
605     ctemp[i] = PREYIN;
606     stemp[i] = ONE;
607   }
608   for (i = NUM_SPECIES/2; i < NUM_SPECIES; i++) {
609     ctemp[i] = PREDIN;
610     stemp[i] = RCONST(0.00001);
611   }
612
613   /* Load initial profiles into cc and sc vector from ctemp and stemp. */
614   for (jy = 0; jy < MY; jy++) {
615     for (jx = 0; jx < MX; jx++) {
616       cloc = IJ_Vptr(cc,jx,jy);
617       sloc = IJ_Vptr(sc,jx,jy);
618       for (i = 0; i < NUM_SPECIES; i++) {
619         cloc[i] = ctemp[i];
620         sloc[i] = stemp[i];
621       }
622     }
623   }
624 }
625
626 /*
627  * Print first lines of output (problem description)
628  */
629
630 static void PrintHeader(int globalstrategy, int maxl, int maxlrst,
631                         realtype fnormtol, realtype scsteptol)
632 {
633   printf("\nPredator-prey test problem --  KINSol (serial version)\n\n");
634   printf("Mesh dimensions = %d X %d\n", MX, MY);
635   printf("Number of species = %d\n", NUM_SPECIES);
636   printf("Total system size = %d\n\n", NEQ);
637   printf("Flag globalstrategy = %d (1 = Inex. Newton, 2 = Linesearch)\n",
638          globalstrategy);
639   printf("Linear solver is SPGMR with maxl = %d, maxlrst = %d\n",
640          maxl, maxlrst);
641   printf("Preconditioning uses interaction-only block-diagonal matrix\n");
642   printf("Positivity constraints imposed on all components \n");
643 #if defined(SUNDIALS_EXTENDED_PRECISION)
644   printf("Tolerance parameters:  fnormtol = %Lg   scsteptol = %Lg\n",
645          fnormtol, scsteptol);
646 #elif defined(SUNDIALS_DOUBLE_PRECISION)
```

```
  printf("Tolerance parameters:  fnormtol = %lg   scsteptol = %lg\n",
         fnormtol, scsteptol);
#else
  printf("Tolerance parameters:  fnormtol = %g   scsteptol = %g\n",
         fnormtol, scsteptol);
#endif

  printf("\nInitial profile of concentration\n");
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("At all mesh points:  %Lg %Lg %Lg   %Lg %Lg %Lg\n",
         PREYIN, PREYIN, PREYIN,
         PREDIN, PREDIN, PREDIN);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("At all mesh points:  %lg %lg %lg   %lg %lg %lg\n",
         PREYIN, PREYIN, PREYIN,
         PREDIN, PREDIN, PREDIN);
#else
  printf("At all mesh points:  %g %g %g   %g %g %g\n",
         PREYIN, PREYIN, PREYIN,
         PREDIN, PREDIN, PREDIN);
#endif
}

/*
 * Print sampled values of current cc
 */

static void PrintOutput(N_Vector cc)
{
  int is, jx, jy;
  realtype *ct;

  jy = 0; jx = 0;
  ct = IJ_Vptr(cc,jx,jy);
  printf("\nAt bottom left:");

  /* Print out lines with up to 6 values per line */
  for (is = 0; is < NUM_SPECIES; is++){
    if ((is%6)*6 == is) printf("\n");
#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf(" %Lg",ct[is]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
    printf(" %lg",ct[is]);
#else
    printf(" %g",ct[is]);
#endif
  }

  jy = MY-1; jx = MX-1;
  ct = IJ_Vptr(cc,jx,jy);
  printf("\n\nAt top right:");

  /* Print out lines with up to 6 values per line */
  for (is = 0; is < NUM_SPECIES; is++) {
```

```
701       if ((is%6)*6 == is) printf("\n");
702   #if defined(SUNDIALS_EXTENDED_PRECISION)
703       printf(" %Lg",ct[is]);
704   #elif defined(SUNDIALS_DOUBLE_PRECISION)
705       printf(" %lg",ct[is]);
706   #else
707       printf(" %g",ct[is]);
708   #endif
709     }
710     printf("\n\n");
711   }
712
713   /*
714    * Print final statistics contained in iopt
715    */
716
717   static void PrintFinalStats(void *kmem)
718   {
719     long int nni, nfe, nli, npe, nps, ncfl, nfeSG;
720     int flag;
721
722     flag = KINGetNumNonlinSolvIters(kmem, &nni);
723     check_flag(&flag, "KINGetNumNonlinSolvIters", 1);
724     flag = KINGetNumFuncEvals(kmem, &nfe);
725     check_flag(&flag, "KINGetNumFuncEvals", 1);
726     flag = KINSpgmrGetNumLinIters(kmem, &nli);
727     check_flag(&flag, "KINSpgmrGetNumLinIters", 1);
728     flag = KINSpgmrGetNumPrecEvals(kmem, &npe);
729     check_flag(&flag, "KINSpgmrGetNumPrecEvals", 1);
730     flag = KINSpgmrGetNumPrecSolves(kmem, &nps);
731     check_flag(&flag, "KINSpgmrGetNumPrecSolves", 1);
732     flag = KINSpgmrGetNumConvFails(kmem, &ncfl);
733     check_flag(&flag, "KINSpgmrGetNumConvFails", 1);
734     flag = KINSpgmrGetNumFuncEvals(kmem, &nfeSG);
735     check_flag(&flag, "KINSpgmrGetNumFuncEvals", 1);
736
737     printf("\nFinal Statistics.. \n\n");
738     printf("nni    = %5ld    nli   = %5ld\n", nni, nli);
739     printf("nfe    = %5ld    nfeSG = %5ld\n", nfe, nfeSG);
740     printf("nps    = %5ld    npe   = %5ld     ncfl  = %5ld\n", nps, npe, ncfl);
741
742   }
743
744   /*
745    * Check function return value...
746    *    opt == 0 means SUNDIALS function allocates memory so check if
747    *             returned NULL pointer
748    *    opt == 1 means SUNDIALS function returns a flag so check if
749    *             flag >= 0
750    *    opt == 2 means function allocates memory so check if returned
751    *             NULL pointer
752    */
753
754   static int check_flag(void *flagvalue, char *funcname, int opt)
```

```
{
  int *errflag;

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

  /* Check if flag < 0 */
  else if (opt == 1) {
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr,
              "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
              funcname, *errflag);
      return(1);
    }
  }

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

  return(0);
}
```

B Listing of kinwebbbd.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.18.2.1 $
 * $Date: 2005/03/17 22:50:58 $
 * -----------------------------------------------------------------
 * Programmer(s): Allan Taylor, Alan Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example problem for KINSol (parallel machine case) using the BBD
 * preconditioner.
 *
 * This example solves a nonlinear system that arises from a system
 * of partial differential equations. 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 the following:
 *
 *       1   2         ns
 * c = (c , c ,  ..., c  )     (denoted by the variable cc)
 *
 * and the PDE's are as follows:
 *
 *                     i       i
 *        0 = d(i)*(c     + c    )  +  f  (x,y,c)   (i=1,...,ns)
 *                     xx      yy         i
 *
 *   where
 *
 *                   i             ns         j
 *   f  (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 number np is both the
 * number of prey and predator species. 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)
 *   b(i) = BB * (1 + alpha * x * y)   (i <= np)
 *   b(i) =-BB * (1 + alpha * x * y)   (i >  np)
 *   d(i) = DPREY   (i <= np)
 *   d(i) = DPRED   ( i > np)
 *
 * The various scalar parameters are set using define's or in
 * routine InitUserData.
 *
 * The boundary conditions are: normal derivative = 0, and the
 * initial guess is constant in x and y, although the final
 * solution is not.
 *
```

```
 * The PDEs are discretized by central differencing on a MX by
 * MY mesh.
 *
 * The nonlinear system is solved by KINSOL using the method
 * specified in the local variable globalstrat.
 *
 * The preconditioner matrix is a band-block-diagonal matrix
 * using the KINBBDPRE module. The half-bandwidths are:
 *
 *   ml = mu = 2*ns - 1
 * ----------------------------------------------------------------
 * References:
 *
 * 1. Peter N. Brown and Youcef Saad,
 *    Hybrid Krylov Methods for Nonlinear Systems of Equations
 *    LLNL report UCRL-97645, November 1987.
 *
 * 2. Peter N. Brown and Alan C. Hindmarsh,
 *    Reduced Storage Matrix Methods in Stiff ODE systems,
 *    Lawrence Livermore National Laboratory Report  UCRL-95088,
 *    Rev. 1, June 1987, and  Journal of Applied Mathematics and
 *    Computation, Vol. 31 (May 1989), pp. 40-91. (Presents a
 *    description of the time-dependent version of this
 *    test problem.)
 * ----------------------------------------------------------------
 *  Run command line: mpirun -np N -machinefile machines kinwebbbd
 *  where N = NPEX * NPEY is the number of processors.
 * ----------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "sundialstypes.h"     /* def's of realtype and booleantype           */
#include "kinsol.h"            /* main KINSol header file                     */
#include "iterative.h"         /* enum for types of preconditioning           */
#include "kinspgmr.h"          /* use KINSpgmr linear solver                  */
#include "smalldense.h"        /* use generic DENSE solver for preconditioning*/
#include "nvector_parallel.h"  /* def's of type N_Vector, macro NV_DATA_P     */
#include "sundialsmath.h"      /* contains RSqrt routine                      */
#include "mpi.h"               /* MPI include file                            */
#include "kinbbdpre.h"         /* band preconditioner function prototypes     */

/* Problem Constants */

#define NUM_SPECIES     6  /*  must equal 2*(number of prey or predators)
                               number of prey =  number of predators      */

#define PI       RCONST(3.1415926535898)   /* pi */

#define NPEX        2           /* number of processors in the x-direction */
#define NPEY        2           /* number of processors in the y-direction */
#define MXSUB       10          /* number of x mesh points per subgrid     */
#define MYSUB       10          /* number of y mesh points per subgrid     */
```

```
#define MX          (NPEX*MXSUB) /* number of grid points in x-direction    */
#define MY          (NPEY*MYSUB) /* number of grid points in y-direction    */
#define NSMXSUB     (NUM_SPECIES * MXSUB)
#define NSMXSUB2    (NUM_SPECIES * (MXSUB+2))
#define NEQ         (NUM_SPECIES*MX*MY)  /* number of equations in system   */
#define AA          RCONST(1.0)    /* value of coefficient AA in above eqns */
#define EE          RCONST(10000.) /* value of coefficient EE in above eqns */
#define GG          RCONST(0.5e-6) /* value of coefficient GG in above eqns */
#define BB          RCONST(1.0)    /* value of coefficient BB in above eqns */
#define DPREY       RCONST(1.0)    /* value of coefficient dprey above */
#define DPRED       RCONST(0.5)    /* value of coefficient dpred above */
#define ALPHA       RCONST(1.0)    /* value of coefficient alpha above */
#define AX          RCONST(1.0)    /* total range of x variable */
#define AY          RCONST(1.0)    /* total range of y variable */
#define FTOL        RCONST(1.e-7)  /*  ftol tolerance */
#define STOL        RCONST(1.e-13) /*  stol tolerance */
#define THOUSAND    RCONST(1000.0) /* one thousand */
#define ZERO        RCONST(0.0)    /* 0. */
#define ONE         RCONST(1.0)    /* 1. */
#define PREYIN      RCONST(1.0)    /* initial guess for prey concentrations. */
#define PREDIN      RCONST(30000.0)/* initial guess for predator concs.      */

/* User-defined vector access macro: IJ_Vptr */

/* IJ_Vptr is defined in order to translate from the underlying 3D structure
   of the dependent variable vector to the 1D storage scheme for an N-vector.
   IJ_Vptr(vv,i,j) returns a pointer to the location in vv corresponding to
   indices is = 0, jx = i, jy = j.    */

#define IJ_Vptr(vv,i,j)   (&NV_Ith_P(vv, i*NUM_SPECIES + j*NSMXSUB))

/* Type : UserData
   contains preconditioner blocks, pivot arrays, and problem constants */

typedef struct {
  realtype **acoef, *bcoef;
  N_Vector rates;
  realtype *cox, *coy;
  realtype ax, ay, dx, dy;
  long int Nlocal, mx, my, ns, np;
  realtype cext[NUM_SPECIES * (MXSUB+2)*(MYSUB+2)];
  long int my_pe, isubx, isuby, nsmxsub, nsmxsub2;
  MPI_Comm comm;
} *UserData;

/* Function called by the KINSol Solver */

static void func(N_Vector cc, N_Vector fval, void *f_data);

static void ccomm(long int Nlocal, N_Vector cc, void *data);

static void func_local(long int Nlocal, N_Vector cc, N_Vector fval, void *f_data);

/* Private Helper Functions */
```

```
161
162 static UserData AllocUserData(void);
163 static void InitUserData(long int my_pe, long int Nlocal, MPI_Comm comm, UserData data);
164 static void FreeUserData(UserData data);
165 static void SetInitialProfiles(N_Vector cc, N_Vector sc);
166 static void PrintHeader(int globalstrategy, int maxl, int maxlrst,
167                         long int mu, long int ml,
168                         realtype fnormtol, realtype scsteptol);
169 static void PrintOutput(long int my_pe, MPI_Comm comm, N_Vector cc);
170 static void PrintFinalStats(void *kmem);
171 static void WebRate(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
172                     void *f_data);
173 static realtype DotProd(long int size, realtype *x1, realtype *x2);
174 static void BSend(MPI_Comm comm, long int my_pe, long int isubx,
175                   long int isuby, long int dsizex, long int dsizey,
176                   realtype *cdata);
177 static void BRecvPost(MPI_Comm comm, MPI_Request request[], long int my_pe,
178                       long int isubx, long int isuby,
179                       long int dsizex, long int dsizey,
180                       realtype *cext, realtype *buffer);
181 static void BRecvWait(MPI_Request request[], long int isubx,
182                       long int isuby, long int dsizex, realtype *cext,
183                       realtype *buffer);
184 static int check_flag(void *flagvalue, char *funcname, int opt, int id);
185
186 /*
187  *--------------------------------------------------------------------
188  * MAIN PROGRAM
189  *--------------------------------------------------------------------
190  */
191
192 int main(int argc, char *argv[])
193 {
194   MPI_Comm comm;
195   void *kmem, *pdata;
196   UserData data;
197   N_Vector cc, sc, constraints;
198   int globalstrategy;
199   long int Nlocal;
200   realtype fnormtol, scsteptol, dq_rel_uu;
201   int flag, maxl, maxlrst;
202   long int mu, ml;
203   int my_pe, npes, npelast = NPEX*NPEY-1;
204
205   data = NULL;
206   kmem = pdata = NULL;
207   cc = sc = constraints = NULL;
208
209   /* Get processor number and total number of pe's */
210   MPI_Init(&argc, &argv);
211   comm = MPI_COMM_WORLD;
212   MPI_Comm_size(comm, &npes);
213   MPI_Comm_rank(comm, &my_pe);
214
```

```
215   if (npes != NPEX*NPEY) {
216     if (my_pe == 0)
217       printf("\nMPI_ERROR(0): npes=%d is not equal to NPEX*NPEY=%d\n", npes, NPEX*NPEY);
218     return(1);
219   }
220
221   /* Allocate memory, and set problem data, initial values, tolerances */
222
223   /* Set local length */
224   Nlocal = NUM_SPECIES*MXSUB*MYSUB;
225
226   /* Allocate and initialize user data block */
227   data = AllocUserData();
228   if (check_flag((void *)data, "AllocUserData", 2, my_pe)) MPI_Abort(comm, 1);
229   InitUserData(my_pe, Nlocal, comm, data);
230
231   /* Choose global strategy */
232   globalstrategy = KIN_INEXACT_NEWTON;
233
234   /* Allocate and initialize vectors */
235   cc = N_VNew_Parallel(comm, Nlocal, NEQ);
236   if (check_flag((void *)cc, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
237   sc = N_VNew_Parallel(comm, Nlocal, NEQ);
238   if (check_flag((void *)sc, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
239   data->rates = N_VNew_Parallel(comm, Nlocal, NEQ);
240   if (check_flag((void *)data->rates, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
241   constraints = N_VNew_Parallel(comm, Nlocal, NEQ);
242   if (check_flag((void *)constraints, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
243   N_VConst(ZERO, constraints);
244
245   SetInitialProfiles(cc, sc);
246
247   fnormtol = FTOL; scsteptol = STOL;
248
249   /* Call KINCreate/KINMalloc to initialize KINSOL:
250      nvSpec  points to machine environment data
251      A pointer to KINSOL problem memory is returned and stored in kmem. */
252   kmem = KINCreate();
253   if (check_flag((void *)kmem, "KINCreate", 0, my_pe)) MPI_Abort(comm, 1);
254
255   /* Vector cc passed as template vector. */
256   flag = KINMalloc(kmem, func, cc);
257   if (check_flag(&flag, "KINMalloc", 1, my_pe)) MPI_Abort(comm, 1);
258
259   flag = KINSetFdata(kmem, data);
260   if (check_flag(&flag, "KINSetFdata", 1, my_pe)) MPI_Abort(comm, 1);
261
262   flag = KINSetConstraints(kmem, constraints);
263   if (check_flag(&flag, "KINSetConstraints", 1, my_pe)) MPI_Abort(comm, 1);
264
265   flag = KINSetFuncNormTol(kmem, fnormtol);
266   if (check_flag(&flag, "KINSetFuncNormTol", 1, my_pe)) MPI_Abort(comm, 1);
267
268   flag = KINSetScaledStepTol(kmem, scsteptol);
```

```
269   if (check_flag(&flag, "KINSetScaledStepTol", 1, my_pe)) MPI_Abort(comm, 1);
270
271   /* Call KINBBDPrecAlloc to initialize and allocate memory for the
272      band-block-diagonal preconditioner, and specify the local and
273      communication functions func_local and gcomm=NULL (all communication
274      needed for the func_local is already done in func). */
275   dq_rel_uu = ZERO;
276   mu = ml = 2*NUM_SPECIES - 1;
277
278   pdata = KINBBDPrecAlloc(kmem, Nlocal, mu, ml, dq_rel_uu, func_local, NULL);
279   if (check_flag((void *)pdata, "KINBBDPrecAlloc", 0, my_pe))
280     MPI_Abort(comm, 1);
281
282   /* Call KINBBDSpgmr to specify the linear solver KINSPGMR
283      with preconditioner KINBBDPRE */
284   maxl = 20; maxlrst = 2;
285   flag = KINBBDSpgmr(kmem, maxl, pdata);
286   if (check_flag(&flag, "KINBBDSpgmr", 1, my_pe))
287     MPI_Abort(comm, 1);
288
289   flag = KINSpgmrSetMaxRestarts(kmem, maxlrst);
290   if (check_flag(&flag, "KINSpgmrSetMaxRestarts", 1, my_pe))
291     MPI_Abort(comm, 1);
292
293   /* Print out the problem size, solution parameters, initial guess. */
294   if (my_pe == 0)
295     PrintHeader(globalstrategy, maxl, maxlrst, mu, ml, fnormtol, scsteptol);
296
297   /* call KINSol and print output concentration profile */
298   flag = KINSol(kmem,           /* KINSol memory block */
299                 cc,             /* initial guesss on input; solution vector */
300                 globalstrategy, /* global stragegy choice */
301                 sc,             /* scaling vector, for the variable cc */
302                 sc);            /* scaling vector for function values fval */
303   if (check_flag(&flag, "KINSol", 1, my_pe)) MPI_Abort(comm, 1);
304
305   if (my_pe == 0) printf("\n\n\nComputed equilibrium species concentrations:\n");
306   if (my_pe == 0 || my_pe==npelast) PrintOutput(my_pe, comm, cc);
307
308   /* Print final statistics and free memory */
309   if (my_pe == 0)
310     PrintFinalStats(kmem);
311
312   N_VDestroy_Parallel(cc);
313   N_VDestroy_Parallel(sc);
314   N_VDestroy_Parallel(constraints);
315   KINBBDPrecFree(pdata);
316   KINFree(kmem);
317   FreeUserData(data);
318
319   MPI_Finalize();
320
321   return(0);
322 }
```

```
323
324 /* Readability definitions used in other routines below */
325
326 #define acoef  (data->acoef)
327 #define bcoef  (data->bcoef)
328 #define cox    (data->cox)
329 #define coy    (data->coy)
330
331 /*
332  *--------------------------------------------------------------------
333  * FUNCTIONS CALLED BY KINSOL
334  *--------------------------------------------------------------------
335  */
336
337 /*
338  * ccomm routine.  This routine performs all communication
339  * between processors of data needed to calculate f.
340  */
341
342 static void ccomm(long int Nlocal, N_Vector cc, void *userdata)
343 {
344
345   realtype *cdata, *cext, buffer[2*NUM_SPECIES*MYSUB];
346   UserData data;
347   MPI_Comm comm;
348   long int my_pe, isubx, isuby, nsmxsub, nsmysub;
349   MPI_Request request[4];
350
351   /* Get comm, my_pe, subgrid indices, data sizes, extended array cext */
352   data = (UserData) userdata;
353   comm = data->comm;  my_pe = data->my_pe;
354   isubx = data->isubx;   isuby = data->isuby;
355   nsmxsub = data->nsmxsub;
356   nsmysub = NUM_SPECIES*MYSUB;
357   cext = data->cext;
358
359   cdata = NV_DATA_P(cc);
360
361   /* Start receiving boundary data from neighboring PEs */
362   BRecvPost(comm, request, my_pe, isubx, isuby, nsmxsub, nsmysub, cext, buffer);
363
364   /* Send data from boundary of local grid to neighboring PEs */
365   BSend(comm, my_pe, isubx, isuby, nsmxsub, nsmysub, cdata);
366
367   /* Finish receiving boundary data from neighboring PEs */
368   BRecvWait(request, isubx, isuby, nsmxsub, cext, buffer);
369
370 }
371
372 /*
373  * System function for predator-prey system - calculation part
374  */
375
376 static void func_local(long int Nlocal, N_Vector cc, N_Vector fval, void *f_data)
```

```
{
  realtype xx, yy, *cxy, *rxy, *fxy, dcydi, dcyui, dcxli, dcxri;
  realtype *cext, dely, delx, *cdata;
  long int i, jx, jy, is, ly;
  long int isubx, isuby, nsmxsub, nsmxsub2;
  long int shifty, offsetc, offsetce, offsetcl, offsetcr, offsetcd, offsetcu;
  UserData data;

  data = (UserData)f_data;
  cdata = NV_DATA_P(cc);

  /* Get subgrid indices, data sizes, extended work array cext */
  isubx = data->isubx;   isuby = data->isuby;
  nsmxsub = data->nsmxsub; nsmxsub2 = data->nsmxsub2;
  cext = data->cext;

  /* Copy local segment of cc vector into the working extended array cext */
  offsetc = 0;
  offsetce = nsmxsub2 + NUM_SPECIES;
  for (ly = 0; ly < MYSUB; ly++) {
    for (i = 0; i < nsmxsub; i++) cext[offsetce+i] = cdata[offsetc+i];
    offsetc = offsetc + nsmxsub;
    offsetce = offsetce + 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 isuby = 0, copy x-line 2 of cc to cext */
  if (isuby == 0) {
    for (i = 0; i < nsmxsub; i++) cext[NUM_SPECIES+i] = cdata[nsmxsub+i];
  }

  /* If isuby = NPEY-1, copy x-line MYSUB-1 of cc to cext */
  if (isuby == NPEY-1) {
    offsetc = (MYSUB-2)*nsmxsub;
    offsetce = (MYSUB+1)*nsmxsub2 + NUM_SPECIES;
    for (i = 0; i < nsmxsub; i++) cext[offsetce+i] = cdata[offsetc+i];
  }

  /* If isubx = 0, copy y-line 2 of cc to cext */
  if (isubx == 0) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetc = ly*nsmxsub + NUM_SPECIES;
      offsetce = (ly+1)*nsmxsub2;
      for (i = 0; i < NUM_SPECIES; i++) cext[offsetce+i] = cdata[offsetc+i];
    }
  }

  /* If isubx = NPEX-1, copy y-line MXSUB-1 of cc to cext */
  if (isubx == NPEX-1) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetc = (ly+1)*nsmxsub - 2*NUM_SPECIES;
      offsetce = (ly+2)*nsmxsub2 - NUM_SPECIES;
```

```
431         for (i = 0; i < NUM_SPECIES; i++) cext[offsetce+i] = cdata[offsetc+i];
432       }
433     }
434 
435     /* Loop over all mesh points, evaluating rate arra at each point */
436     delx = data->dx;
437     dely = data->dy;
438     shifty = (MXSUB+2)*NUM_SPECIES;
439 
440     for (jy = 0; jy < MYSUB; jy++) {
441 
442       yy = dely*(jy + isuby * MYSUB);
443 
444       for (jx = 0; jx < MXSUB; jx++) {
445 
446         xx = delx * (jx + isubx * MXSUB);
447         cxy = IJ_Vptr(cc,jx,jy);
448         rxy = IJ_Vptr(data->rates,jx,jy);
449         fxy = IJ_Vptr(fval,jx,jy);
450 
451         WebRate(xx, yy, cxy, rxy, f_data);
452 
453         offsetc = (jx+1)*NUM_SPECIES + (jy+1)*NSMXSUB2;
454         offsetcd = offsetc - shifty;
455         offsetcu = offsetc + shifty;
456         offsetcl = offsetc - NUM_SPECIES;
457         offsetcr = offsetc + NUM_SPECIES;
458 
459         for (is = 0; is < NUM_SPECIES; is++) {
460 
461           /* differencing in x */
462           dcydi = cext[offsetc+is]  - cext[offsetcd+is];
463           dcyui = cext[offsetcu+is] - cext[offsetc+is];
464 
465           /* differencing in y */
466           dcxli = cext[offsetc+is]  - cext[offsetcl+is];
467           dcxri = cext[offsetcr+is] - cext[offsetc+is];
468 
469           /* compute the value at xx , yy */
470           fxy[is] = (coy)[is] * (dcyui - dcydi) +
471             (cox)[is] * (dcxri - dcxli) + rxy[is];
472 
473         } /* end of is loop */
474 
475       } /* end of jx loop */
476 
477     } /* end of jy loop */
478   }
479 
480   /*
481    * System function routine.  Evaluate f(cc).  First call ccomm to do
482    * communication of subgrid boundary data into cext.  Then calculate f
483    * by a call to func_local.
484    */
```

```
485
486 static void func(N_Vector cc, N_Vector fval, void *f_data)
487 {
488   UserData data;
489
490   data = (UserData) f_data;
491
492   /* Call ccomm to do inter-processor communicaiton */
493   ccomm(data->Nlocal, cc, data);
494
495   /* Call func_local to calculate all right-hand sides */
496   func_local(data->Nlocal, cc, fval, data);
497 }
498
499 /*
500  * Interaction rate function routine
501  */
502
503 static void WebRate(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
504                     void *f_data)
505 {
506   long int i;
507   realtype fac;
508   UserData data;
509
510   data = (UserData)f_data;
511
512   for (i = 0; i<NUM_SPECIES; i++)
513     ratesxy[i] = DotProd(NUM_SPECIES, cxy, acoef[i]);
514
515   fac = ONE + ALPHA * xx * yy;
516
517   for (i = 0; i < NUM_SPECIES; i++)
518     ratesxy[i] = cxy[i] * ( bcoef[i] * fac + ratesxy[i] );
519 }
520
521 /*
522  * Dot product routine for realtype arrays
523  */
524
525 static realtype DotProd(long int size, realtype *x1, realtype *x2)
526 {
527   long int i;
528   realtype *xx1, *xx2, temp = ZERO;
529
530   xx1 = x1; xx2 = x2;
531   for (i = 0; i < size; i++) temp += (*xx1++) * (*xx2++);
532
533   return(temp);
534 }
535
536 /*
537  *-----------------------------------------------------------------
538  * PRIVATE FUNCTIONS
```

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

/*
 * Allocate memory for data structure of type UserData
 */

static UserData AllocUserData(void)
{
  UserData data;

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

  acoef = denalloc(NUM_SPECIES);
  bcoef = (realtype *)malloc(NUM_SPECIES * sizeof(realtype));
  cox   = (realtype *)malloc(NUM_SPECIES * sizeof(realtype));
  coy   = (realtype *)malloc(NUM_SPECIES * sizeof(realtype));

  return(data);
}

/*
 * Load problem constants in data
 */

static void InitUserData(long int my_pe, long int Nlocal, MPI_Comm comm, UserData data)
{
  long int i, j, np;
  realtype *a1,*a2, *a3, *a4, dx2, dy2;

  data->mx = MX;
  data->my = MY;
  data->ns = NUM_SPECIES;
  data->np = NUM_SPECIES/2;
  data->ax = AX;
  data->ay = AY;
  data->dx = (data->ax)/(MX-1);
  data->dy = (data->ay)/(MY-1);
  data->my_pe = my_pe;
  data->Nlocal = Nlocal;
  data->comm = comm;
  data->isuby = my_pe/NPEX;
  data->isubx = my_pe - data->isuby*NPEX;
  data->nsmxsub = NUM_SPECIES * MXSUB;
  data->nsmxsub2 = NUM_SPECIES * (MXSUB+2);

  /* Set up the coefficients a and b plus others found in the equations */
  np = data->np;

  dx2=(data->dx)*(data->dx); dy2=(data->dy)*(data->dy);

  for (i = 0; i < np; i++) {
    a1= &(acoef[i][np]);
    a2= &(acoef[i+np][0]);
```

```
593       a3= &(acoef[i][0]);
594       a4= &(acoef[i+np][np]);
595
596       /*  Fill in the portion of acoef in the four quadrants, row by row */
597       for (j = 0; j < np; j++) {
598         *a1++ =  -GG;
599         *a2++ =   EE;
600         *a3++ = ZERO;
601         *a4++ = ZERO;
602       }
603
604       /* and then change the diagonal elements of acoef to -AA */
605       acoef[i][i]=-AA;
606       acoef[i+np][i+np] = -AA;
607
608       bcoef[i] = BB;
609       bcoef[i+np] = -BB;
610
611       cox[i]=DPREY/dx2;
612       cox[i+np]=DPRED/dx2;
613
614       coy[i]=DPREY/dy2;
615       coy[i+np]=DPRED/dy2;
616     }
617   }
618
619   /*
620    * Free data memory
621    */
622
623   static void FreeUserData(UserData data)
624   {
625
626     denfree(acoef);
627     free(bcoef);
628     free(cox); free(coy);
629     N_VDestroy_Parallel(data->rates);
630
631     free(data);
632
633   }
634
635   /*
636    * Set initial conditions in cc
637    */
638
639   static void SetInitialProfiles(N_Vector cc, N_Vector sc)
640   {
641     int i, jx, jy;
642     realtype *cloc, *sloc;
643     realtype  ctemp[NUM_SPECIES], stemp[NUM_SPECIES];
644
645     /* Initialize arrays ctemp and stemp used in the loading process */
646     for (i = 0; i < NUM_SPECIES/2; i++) {
```

```
647       ctemp[i] = PREYIN;
648       stemp[i] = ONE;
649     }
650     for (i = NUM_SPECIES/2; i < NUM_SPECIES; i++) {
651       ctemp[i] = PREDIN;
652       stemp[i] = RCONST(0.00001);
653     }
654 
655     /* Load initial profiles into cc and sc vector from ctemp and stemp. */
656     for (jy = 0; jy < MYSUB; jy++) {
657       for (jx=0; jx < MXSUB; jx++) {
658         cloc = IJ_Vptr(cc,jx,jy);
659         sloc = IJ_Vptr(sc,jx,jy);
660         for (i = 0; i < NUM_SPECIES; i++){
661           cloc[i] = ctemp[i];
662           sloc[i] = stemp[i];
663         }
664       }
665     }
666 
667   }
668 
669   /*
670    * Print first lines of output (problem description)
671    */
672 
673   static void PrintHeader(int globalstrategy, int maxl, int maxlrst,
674                           long int mu, long int ml,
675                           realtype fnormtol, realtype scsteptol)
676   {
677       printf("\nPredator-prey test problem--  KINSol (parallel-BBD version)\n\n");
678 
679       printf("Mesh dimensions = %d X %d\n", MX, MY);
680       printf("Number of species = %d\n", NUM_SPECIES);
681       printf("Total system size = %d\n\n", NEQ);
682       printf("Subgrid dimensions = %d X %d\n", MXSUB, MYSUB);
683       printf("Processor array is %d X %d\n\n", NPEX, NPEY);
684       printf("Flag globalstrategy = %d (1 = Inex. Newton, 2 = Linesearch)\n",
685              globalstrategy);
686       printf("Linear solver is SPGMR with maxl = %d, maxlrst = %d\n",
687              maxl, maxlrst);
688       printf("Preconditioning uses band-block-diagonal matrix from KINBBDPRE\n");
689       printf("  with matrix half-bandwidths ml, mu = %ld  %ld\n", ml, mu);
690   #if defined(SUNDIALS_EXTENDED_PRECISION)
691       printf("Tolerance parameters:  fnormtol = %Lg   scsteptol = %Lg\n",
692              fnormtol, scsteptol);
693   #elif defined(SUNDIALS_DOUBLE_PRECISION)
694       printf("Tolerance parameters:  fnormtol = %lg   scsteptol = %lg\n",
695              fnormtol, scsteptol);
696   #else
697       printf("Tolerance parameters:  fnormtol = %g   scsteptol = %g\n",
698              fnormtol, scsteptol);
699   #endif
700 
```

```
701       printf("\nInitial profile of concentration\n");
702   #if defined(SUNDIALS_EXTENDED_PRECISION)
703       printf("At all mesh points:  %Lg %Lg %Lg   %Lg %Lg %Lg\n", PREYIN,PREYIN,PREYIN,
704              PREDIN,PREDIN,PREDIN);
705   #elif defined(SUNDIALS_DOUBLE_PRECISION)
706       printf("At all mesh points:  %lg %lg %lg   %lg %lg %lg\n", PREYIN,PREYIN,PREYIN,
707              PREDIN,PREDIN,PREDIN);
708   #else
709       printf("At all mesh points:  %g %g %g   %g %g %g\n", PREYIN,PREYIN,PREYIN,
710              PREDIN,PREDIN,PREDIN);
711   #endif
712   }
713
714   /*
715    * Print sample of current cc values
716    */
717
718   static void PrintOutput(long int my_pe, MPI_Comm comm, N_Vector cc)
719   {
720     int is, i0, npelast;
721     realtype  *ct, tempc[NUM_SPECIES];
722     MPI_Status status;
723
724     npelast = NPEX*NPEY - 1;
725
726     ct = NV_DATA_P(cc);
727
728     /* Send the cc values (for all species) at the top right mesh point to PE 0 */
729     if (my_pe == npelast) {
730       i0 = NUM_SPECIES*(MXSUB*MYSUB-1);
731       if (npelast!=0)
732         MPI_Send(&ct[i0],NUM_SPECIES,PVEC_REAL_MPI_TYPE,0,0,comm);
733       else  /* single processor case */
734         for (is = 0; is < NUM_SPECIES; is++) tempc[is]=ct[i0+is];
735     }
736
737     /* On PE 0, receive the cc values at top right, then print performance data
738        and sampled solution values */
739     if (my_pe == 0) {
740
741       if (npelast != 0)
742         MPI_Recv(&tempc[0],NUM_SPECIES,PVEC_REAL_MPI_TYPE,npelast,0,comm,&status);
743
744       printf("\nAt bottom left:");
745       for (is = 0; is < NUM_SPECIES; is++){
746         if ((is%6)*6== is) printf("\n");
747   #if defined(SUNDIALS_EXTENDED_PRECISION)
748         printf(" %Lg",ct[is]);
749   #elif defined(SUNDIALS_DOUBLE_PRECISION)
750         printf(" %lg",ct[is]);
751   #else
752         printf(" %g",ct[is]);
753   #endif
754       }
```

```
755
756     printf("\n\nAt top right:");
757     for (is = 0; is < NUM_SPECIES; is++) {
758       if ((is%6)*6 == is) printf("\n");
759 #if defined(SUNDIALS_EXTENDED_PRECISION)
760       printf(" %Lg",tempc[is]);
761 #elif defined(SUNDIALS_DOUBLE_PRECISION)
762       printf(" %lg",tempc[is]);
763 #else
764       printf(" %g",tempc[is]);
765 #endif
766     }
767     printf("\n\n");
768   }
769 }
770
771 /*
772  * Print final statistics contained in iopt
773  */
774
775 static void PrintFinalStats(void *kmem)
776 {
777   long int nni, nfe, nli, npe, nps, ncfl, nfeSG;
778   int flag;
779
780   flag = KINGetNumNonlinSolvIters(kmem, &nni);
781   check_flag(&flag, "KINGetNumNonlinSolvIters", 1, 0);
782   flag = KINGetNumFuncEvals(kmem, &nfe);
783   check_flag(&flag, "KINGetNumFuncEvals", 1, 0);
784   flag = KINSpgmrGetNumLinIters(kmem, &nli);
785   check_flag(&flag, "KINSpgmrGetNumLinIters", 1, 0);
786   flag = KINSpgmrGetNumPrecEvals(kmem, &npe);
787   check_flag(&flag, "KINSpgmrGetNumPrecEvals", 1, 0);
788   flag = KINSpgmrGetNumPrecSolves(kmem, &nps);
789   check_flag(&flag, "KINSpgmrGetNumPrecSolves", 1, 0);
790   flag = KINSpgmrGetNumConvFails(kmem, &ncfl);
791   check_flag(&flag, "KINSpgmrGetNumConvFails", 1, 0);
792   flag = KINSpgmrGetNumFuncEvals(kmem, &nfeSG);
793   check_flag(&flag, "KINSpgmrGetNumFuncEvals", 1, 0);
794
795   printf("\nFinal Statistics.. \n\n");
796   printf("nni    = %5ld    nli   = %5ld\n", nni, nli);
797   printf("nfe    = %5ld    nfeSG = %5ld\n", nfe, nfeSG);
798   printf("nps    = %5ld    npe   = %5ld     ncfl  = %5ld\n", nps, npe, ncfl);
799
800 }
801
802 /*
803  * Routine to send boundary data to neighboring PEs
804  */
805
806 static void BSend(MPI_Comm comm, long int my_pe,
807                   long int isubx, long int isuby,
808                   long int dsizex, long int dsizey, realtype *cdata)
```

```
{
  int i, ly;
  long int offsetc, offsetbuf;
  realtype bufleft[NUM_SPECIES*MYSUB], bufright[NUM_SPECIES*MYSUB];

  /* If isuby > 0, send data from bottom x-line of u */
  if (isuby != 0)
    MPI_Send(&cdata[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);

  /* If isuby < NPEY-1, send data from top x-line of u */
  if (isuby != NPEY-1) {
    offsetc = (MYSUB-1)*dsizex;
    MPI_Send(&cdata[offsetc], dsizex, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
  }

  /* If isubx > 0, send data from left y-line of u (via bufleft) */
  if (isubx != 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 isubx < NPEX-1, send data from right y-line of u (via bufright) */
  if (isubx != 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);
  }
}

/*
 * Routine to start receiving boundary data from neighboring PEs.
 *  Notes:
 *  1) buffer should be able to hold 2*NUM_SPECIES*MYSUB realtype 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 should be passed in both calls also.
 */

static void BRecvPost(MPI_Comm comm, MPI_Request request[], long int my_pe,
                      long int isubx, long int isuby,
                      long int dsizex, long int dsizey,
                      realtype *cext, realtype *buffer)
{
  long int offsetce;

```

```
863   /* Have bufleft and bufright use the same buffer */
864   realtype *bufleft = buffer, *bufright = buffer+NUM_SPECIES*MYSUB;
865
866   /* If isuby > 0, receive data for bottom x-line of cext */
867   if (isuby != 0)
868     MPI_Irecv(&cext[NUM_SPECIES], dsizex, PVEC_REAL_MPI_TYPE,
869               my_pe-NPEX, 0, comm, &request[0]);
870
871   /* If isuby < NPEY-1, receive data for top x-line of cext */
872   if (isuby != NPEY-1) {
873     offsetce = NUM_SPECIES*(1 + (MYSUB+1)*(MXSUB+2));
874     MPI_Irecv(&cext[offsetce], dsizex, PVEC_REAL_MPI_TYPE,
875               my_pe+NPEX, 0, comm, &request[1]);
876   }
877
878   /* If isubx > 0, receive data for left y-line of cext (via bufleft) */
879   if (isubx != 0) {
880     MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
881               my_pe-1, 0, comm, &request[2]);
882   }
883
884   /* If isubx < NPEX-1, receive data for right y-line of cext (via bufright) */
885   if (isubx != NPEX-1) {
886     MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
887               my_pe+1, 0, comm, &request[3]);
888   }
889 }
890
891 /*
892  * Routine to finish receiving boundary data from neighboring PEs.
893  *  Notes:
894  *  1) buffer should be able to hold 2*NUM_SPECIES*MYSUB realtype entries,
895  *     should be passed to both the BRecvPost and BRecvWait functions, and
896  *     should not be manipulated between the two calls.
897  *  2) request should have 4 entries, and should be passed in both calls also.
898  */
899
900 static void BRecvWait(MPI_Request request[], long int isubx,
901                       long int isuby, long int dsizex, realtype *cext,
902                       realtype *buffer)
903 {
904   int i, ly;
905   long int dsizex2, offsetce, offsetbuf;
906   realtype *bufleft = buffer, *bufright = buffer+NUM_SPECIES*MYSUB;
907   MPI_Status status;
908
909   dsizex2 = dsizex + 2*NUM_SPECIES;
910
911   /* If isuby > 0, receive data for bottom x-line of cext */
912   if (isuby != 0)
913     MPI_Wait(&request[0],&status);
914
915   /* If isuby < NPEY-1, receive data for top x-line of cext */
916   if (isuby != NPEY-1)
```

```
917       MPI_Wait(&request[1],&status);
918
919     /* If isubx > 0, receive data for left y-line of cext (via bufleft) */
920     if (isubx != 0) {
921       MPI_Wait(&request[2],&status);
922
923       /* Copy the buffer to cext */
924       for (ly = 0; ly < MYSUB; ly++) {
925         offsetbuf = ly*NUM_SPECIES;
926         offsetce = (ly+1)*dsizex2;
927         for (i = 0; i < NUM_SPECIES; i++)
928           cext[offsetce+i] = bufleft[offsetbuf+i];
929       }
930     }
931
932     /* If isubx < NPEX-1, receive data for right y-line of cext (via bufright) */
933     if (isubx != NPEX-1) {
934       MPI_Wait(&request[3],&status);
935
936       /* Copy the buffer to cext */
937       for (ly = 0; ly < MYSUB; ly++) {
938         offsetbuf = ly*NUM_SPECIES;
939         offsetce = (ly+2)*dsizex2 - NUM_SPECIES;
940         for (i = 0; i < NUM_SPECIES; i++)
941           cext[offsetce+i] = bufright[offsetbuf+i];
942       }
943     }
944   }
945   /*
946    * Check function return value...
947    *    opt == 0 means SUNDIALS function allocates memory so check if
948    *             returned NULL pointer
949    *    opt == 1 means SUNDIALS function returns a flag so check if
950    *             flag >= 0
951    *    opt == 2 means function allocates memory so check if returned
952    *             NULL pointer
953    */
954
955   static int check_flag(void *flagvalue, char *funcname, int opt, int id)
956   {
957     int *errflag;
958
959     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
960     if (opt == 0 && flagvalue == NULL) {
961       fprintf(stderr,
962               "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
963               id, funcname);
964       return(1);
965     }
966
967     /* Check if flag < 0 */
968     else if (opt == 1) {
969       errflag = (int *) flagvalue;
970       if (*errflag < 0) {
```

```
971         fprintf(stderr,
972                 "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
973                 id, funcname, *errflag);
974         return(1);
975       }
976     }
977
978     /* Check if function returned NULL pointer - no memory allocated */
979     else if (opt == 2 && flagvalue == NULL) {
980       fprintf(stderr,
981               "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
982               id, funcname);
983       return(1);
984     }
985
986     return(0);
987   }
```

C Listing of kindiagsf.f

```
      program kindiagsf
c     ----------------------------------------------------------------
c     $Revision: 1.13 $
c     $Date: 2004/10/15 00:03:54 $
c     ----------------------------------------------------------------
c     Programmer(s): Allan Taylor, Alan Hindmarsh and
c                    Radu Serban @ LLNL
c     ----------------------------------------------------------------
c     Simple diagonal test with Fortran interface, using user-supplied
c     preconditioner setup and solve routines (supplied in Fortran).
c
c     This example does a basic test of the solver by solving the
c     system:
c               f(u) = 0  for
c               f(u) = u(i)^2 - i^2
c
c     No scaling is done.
c     An approximate diagonal preconditioner is used.
c
c     Execution command: kindiagsf
c     ----------------------------------------------------------------
c
      implicit none

      integer ier, globalstrat, inopt, maxl, maxlrst
      integer*4 PROBSIZE
      parameter(PROBSIZE=128)
      integer*4 neq, i, msbpre
      integer*4 iopt(40)
      double precision pp, fnormtol, scsteptol
      double precision ropt(40), uu(PROBSIZE), scale(PROBSIZE)
      double precision constr(PROBSIZE)

      common /pcom/ pp(PROBSIZE)
      common /psize/ neq

      neq = PROBSIZE
      globalstrat = 1
      fnormtol = 1.0d-5
      scsteptol = 1.0d-4
      inopt = 0
      maxl = 10
      maxlrst = 2
      msbpre  = 5

c * * * * * * * * * * * * * * * * * * * * * * *

      call fnvinits(neq, ier)
      if (ier .ne. 0) then
         write(6,1220) ier
 1220    format('SUNDIALS_ERROR: FNVINITS returned IER = ', i2)
         stop
```

```
      endif

      do 20 i = 1, neq
         uu(i) = 2.0d0 * i
         scale(i) = 1.0d0
         constr(i) = 0.0d0
 20   continue

      call fkinmalloc(msbpre, fnormtol, scsteptol,
     &                constr, inopt, iopt, ropt, ier)
      if (ier .ne. 0) then
         write(6,1230) ier
 1230    format('SUNDIALS_ERROR: FKINMALLOC returned IER = ', i2)
         call fnvfrees
         stop
      endif

      call fkinspgmr(maxl, maxlrst, ier)
      if (ier .ne. 0) then
         write(6,1235) ier
 1235    format('SUNDIALS_ERROR: FKINSPGMR returned IER = ', i2)
         call fkinfree
         call fnvfrees
         stop
      endif

      call fkinspgmrsetpsol(1, ier)
      call fkinspgmrsetpset(1, ier)

      write(6,1240)
 1240 format('Example program kindiagsf:'//' This fkinsol example code',
     1       ' solves a 128 eqn diagonal algebraic system.'/
     2       ' Its purpose is to demonstrate the use of the Fortran',
     3       ' interface'/' in a serial environment.'///
     4       ' globalstrategy = KIN_INEXACT_NEWTON')

      call fkinsol(uu, globalstrat, scale, scale, ier)
      if (ier .lt. 0) then
         write(6,1242) ier, iopt(15)
 1242    format('SUNDIALS_ERROR: FKINSOL returned IER = ', i2, /,
     1          '                Linear Solver returned IER = ', i2)
         call fkinfree
         call fnvfrees
         stop
      endif

      write(6,1245) ier
 1245 format(/' FKINSOL return code is ', i3)

      write(6,1246)
 1246 format(//' The resultant values of uu are:'/)

      do 30 i = 1, neq, 4
         write(6,1256) i, uu(i), uu(i+1), uu(i+2), uu(i+3)
```

```
 1256    format(i4, 4(1x, f10.6))
 30   continue

      write(6,1267) iopt(4), iopt(11), iopt(5), iopt(12), iopt(13),
     1              iopt(14)
 1267 format(//'Final statistics:'//
     1       ' nni = ', i4, ',  nli = ', i4, ',  nfe = ', i4,
     2       ',  npe = ', i4, ',  nps = ', i4, ',  ncfl = ', i4)

      call fkinfree
      call fnvfrees

      stop
      end

c * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
c     The function defining the system f(u) = 0 must be defined by a Fortran
c     function of the following form.

      subroutine fkfun(uu, fval)

      implicit none

      integer*4 neq, i
      double precision fval(*), uu(*)

      common /psize/ neq

      do 10 i = 1, neq
         fval(i) = uu(i) * uu(i) - i * i
 10   continue
      return
      end


c * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
c     The routine kpreco is the preconditioner setup routine. It must have
c     that specific name be used in order that the c code can find and link
c     to it.  The argument list must also be as illustrated below:

      subroutine fkpset(udata, uscale, fdata, fscale,
     1                  vtemp1, vtemp2, ier)

      implicit none

      integer ier
      integer*4 neq, i
      double precision pp
      double precision udata(*), uscale(*), fdata(*), fscale(*)
      double precision vtemp1(*), vtemp2(*)

      common /pcom/ pp(128)
      common /psize/ neq

```

```
      do 10 i = 1, neq
         pp(i) = 0.5d0 / (udata(i) + 5.0d0)
 10   continue
      ier = 0

      return
      end


c * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
c     The routine kpsol is the preconditioner solve routine. It must have
c     that specific name be used in order that the c code can find and link
c     to it.  The argument list must also be as illustrated below:

      subroutine fkpsol(udata, uscale, fdata, fscale,
     1                  vv, ftem, ier)

      implicit none

      integer ier
      integer*4 neq, i
      double precision pp
      double precision udata(*), uscale(*), fdata(*), fscale(*)
      double precision vv(*), ftem(*)

      common /pcom/ pp(128)
      common /psize/ neq

      do 10 i = 1, neq
         vv(i) = vv(i) * pp(i)
 10   continue
      ier = 0

      return
      end
```

D Listing of `kindiagpf.f`

```
      program kindiagpf
c     ----------------------------------------------------------------
c     $Revision: 1.13 $
c     $Date: 2004/10/15 00:03:58 $
c     ----------------------------------------------------------------
c     Programmer(s): Allan G. Taylor, Alan C. Hindmarsh and
c                    Radu Serban @ LLNL
c     ----------------------------------------------------------------
c     Simple diagonal test with Fortran interface, using
c     user-supplied preconditioner setup and solve routines (supplied
c     in Fortran, below).
c
c     This example does a basic test of the solver by solving the
c     system:
c               f(u) = 0  for
c               f(u) = u(i)^2 - i^2
c
c      No scaling is done.
c      An approximate diagonal preconditioner is used.
c
c      Execution command: mpirun -np 4 kindiagpf
c     ----------------------------------------------------------------
c
      include "mpif.h"

      integer ier, size, globalstrat, rank, inopt, mype, npes
      integer maxl, maxlrst
      integer*4 localsize
      parameter(localsize=32)
      integer*4 neq, nlocal, msbpre, baseadd, i, ii
      integer*4 iopt(40)
      double precision pp, fnormtol, scsteptol
      double precision uu(localsize), scale(localsize)
      double precision constr(localsize)

      common /pcom/ pp(localsize), mype, npes, baseadd, nlocal

      nlocal = localsize
      neq = 4 * nlocal
      globalstrat = 1
      fnormtol = 1.0d-5
      scsteptol = 1.0d-4
      inopt = 0
      maxl = 10
      maxlrst = 2
      msbpre  = 5

c     The user MUST call mpi_init, Fortran binding, for the fkinsol package
c     to work. The communicator, MPI_COMM_WORLD, is the only one common
c     between the Fortran and C bindings. So in the following, the communicator
c     MPI_COMM_WORLD is used in calls to mpi_comm_size and mpi_comm_rank
c     to determine the total number of processors and the rank (0 ... size-1)
```

```
c     number of this process.

      call mpi_init(ier)
      if (ier .ne. 0) then
         write(6,1210) ier
 1210    format('MPI_ERROR: MPI_INIT returned IER = ', i2)
         stop
      endif

      call fnvinitp(nlocal, neq, ier)
      if (ier .ne. 0) then
         write(6,1220) ier
 1220    format('SUNDIALS_ERROR: FNVINITP returned IER = ', i2)
         call mpi_finalize(ier)
         stop
      endif

      call mpi_comm_size(MPI_COMM_WORLD, size, ier)
      if (ier .ne. 0) then
         write(6,1222) ier
 1222    format('MPI_ERROR: MPI_COMM_SIZE returned IER = ', i2)
         call mpi_abort(MPI_COMM_WORLD, 1, ier)
         stop
      endif

      if (size .ne. 4) then
         write(6,1230)
 1230    format('MPI_ERROR: must use 4 processes')
         call mpi_finalize(ier)
         stop
      endif
      npes = size

      call mpi_comm_rank(MPI_COMM_WORLD, rank, ier)
      if (ier .ne. 0) then
         write(6,1224) ier
 1224    format('MPI_ERROR: MPI_COMM_RANK returned IER = ', i2)
         call mpi_abort(MPI_COMM_WORLD, 1, ier)
         stop
      endif

      mype = rank
      baseadd = mype * nlocal

      do 20 ii = 1, nlocal
         i = ii + baseadd
         uu(ii) = 2.0d0 * i
         scale(ii) = 1.0d0
         constr(ii) = 0.0d0
 20   continue

      call fkinmalloc(msbpre, fnormtol, scsteptol,
     &                constr, inopt, iopt, ropt, ier)

```

```
      if (ier .ne. 0) then
         write(6,1231)ier
 1231    format('SUNDIALS_ERROR: FKINMALLOC returned IER = ', i2)
         call mpi_abort(MPI_COMM_WORLD, 1, ier)
         stop
      endif

      call fkinspgmr(maxl, maxlrst, ier)
      call fkinspgmrsetpsol(1, ier)
      call fkinspgmrsetpset(1, ier)

      if (mype .eq. 0) write(6,1240)
 1240 format('Example program kindiagpf:'//' This fkinsol example code',
     1       ' solves a 128 eqn diagonal algebraic system.'/
     2       ' Its purpose is to demonstrate the use of the Fortran',
     3       ' interface'/' in a parallel environment.'///
     4       ' globalstrategy = KIN_INEXACT_NEWTON')

      call fkinsol(uu, globalstrat, scale, scale, ier)
      if (ier .lt. 0) then
         write(6,1242) ier, iopt(15)
 1242    format('SUNDIALS_ERROR: FKINSOL returned IER = ', i2, /,
     1          '                Linear Solver returned IER = ', i2)
         call mpi_abort(MPI_COMM_WORLD, 1, ier)
         stop
      endif

      if (mype .eq. 0) write(6,1245) ier
 1245 format(/' FKINSOL return code is ', i4/)

      if (mype .eq. 0) write(6,1246)
 1246 format(/' The resultant values of uu (process 0) are:'/)

      do 30 i = 1, nlocal, 4
         if(mype .eq. 0) write(6,1256) i + baseadd, uu(i), uu(i+1),
     1                                 uu(i+2), uu(i+3)
 1256    format(i4, 4(1x, f10.6))
 30   continue

      if (mype .eq. 0) write(6,1267) iopt(4), iopt(11), iopt(5),
     1                               iopt(12), iopt(13), iopt(14)
 1267 format(//'Final statistics:'//
     1       ' nni = ', i4, ',  nli = ', i4, ',  nfe = ', i4,
     2       ',  npe = ', i4, ',  nps=', i4, ',  ncfl=', i4)

      call fkinfree
      call fnvfreep

c     An explicit call to mpi_finalize (Fortran binding) is required by
c     the constructs used in fkinsol.
      call mpi_finalize(ier)

      stop
      end
```

```
161
162
163   c * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
164   c     The function defining the system f(u) = 0 must be defined by a Fortran
165   c     function with the following name and form.
166
167         subroutine fkfun(uu, fval)
168
169         implicit none
170
171         integer mype, npes
172         integer*4 baseadd, nlocal, i, localsize
173         parameter(localsize=32)
174         double precision pp
175         double precision fval(*), uu(*)
176
177         common /pcom/ pp(localsize), mype, npes, baseadd, nlocal
178
179         do 10 i = 1, nlocal
180    10      fval(i) = uu(i) * uu(i) - (i + baseadd) * (i + baseadd)
181
182         return
183         end
184
185
186   c * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
187   c     The routine kpreco is the preconditioner setup routine. It must have
188   c     that specific name be used in order that the c code can find and link
189   c     to it.  The argument list must also be as illustrated below:
190
191         subroutine fkpset(udata, uscale, fdata, fscale,
192        1                  vtemp1, vtemp2, ier)
193
194         implicit none
195
196         integer ier, mype, npes
197         integer*4 localsize
198         parameter(localsize=32)
199         integer*4 baseadd, nlocal, i
200         double precision pp
201         double precision udata(*), uscale(*), fdata(*), fscale(*)
202         double precision vtemp1(*), vtemp2(*)
203
204         common /pcom/ pp(localsize), mype, npes, baseadd, nlocal
205
206         do 10 i = 1, nlocal
207    10      pp(i) = 0.5d0 / (udata(i)+ 5.0d0)
208
209         ier = 0
210
211         return
212         end
213
214
```

```
c * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
c     The routine kpsol is the preconditioner solve routine. It must have
c     that specific name be used in order that the c code can find and link
c     to it.  The argument list must also be as illustrated below:

      subroutine fkpsol(udata, uscale, fdata, fscale,
     1                  vv, ftem, ier)

      implicit none

      integer ier, mype, npes
      integer*4 baseadd, nlocal, i
      integer*4 localsize
      parameter(localsize=32)
      double precision udata(*), uscale(*), fdata(*), fscale(*)
      double precision vv(*), ftem(*)
      double precision pp

      common /pcom/ pp(localsize), mype, npes, baseadd, nlocal

      do 10 i = 1, nlocal
 10      vv(i) = vv(i) * pp(i)

      ier = 0

      return
      end
```