Example Programs for KINSOL v2.3.0

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

April 2005

UCRL-SM-208114

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, and the pointer to user data are specified through a call to KINSpgmrSetPreconditioner (see §5.4.4). The data pointer passed to KINSpgmrSetPreconditioner 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 `fkinspgmrsetprec` routine 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: 1.14.2.3 $
 * $Date: 2005/04/07 00:15:48 $
 * -----------------------------------------------------------------
 * 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);
```

```
static void FreeUserData(UserData data);
static void SetInitialProfiles(N_Vector cc, N_Vector sc);
static void PrintHeader(int globalstrategy, int maxl, int maxlrst,
                        realtype fnormtol, realtype scsteptol);
static void PrintOutput(N_Vector cc);
static void PrintFinalStats(void *kmem);
static void WebRate(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
                    void *f_data);
static realtype DotProd(long int size, realtype *x1, realtype *x2);
static int check_flag(void *flagvalue, char *funcname, int opt);

/*
 *-----------------------------------------------------------------
 * MAIN PROGRAM
 *-----------------------------------------------------------------
 */

int main(void)
{
  int globalstrategy;
  realtype fnormtol, scsteptol;
  N_Vector cc, sc, constraints;
  UserData data;
  int flag, maxl, maxlrst;
  void *kmem;

  cc = sc = constraints = NULL;
  kmem = NULL;
  data = NULL;

  /* Allocate memory, and set problem data, initial values, tolerances */
  globalstrategy = KIN_INEXACT_NEWTON;

  data = AllocUserData();
  if (check_flag((void *)data, "AllocUserData", 2)) return(1);
  InitUserData(data);

  /* Create serial vectors of length NEQ */
  cc = N_VNew_Serial(NEQ);
  if (check_flag((void *)cc, "N_VNew_Serial", 0)) return(1);
  sc = N_VNew_Serial(NEQ);
  if (check_flag((void *)sc, "N_VNew_Serial", 0)) return(1);
  data->rates = N_VNew_Serial(NEQ);
  if (check_flag((void *)data->rates, "N_VNew_Serial", 0)) return(1);

  constraints = N_VNew_Serial(NEQ);
  if (check_flag((void *)constraints, "N_VNew_Serial", 0)) return(1);
  N_VConst(TWO, constraints);

  SetInitialProfiles(cc, sc);

  fnormtol=FTOL; scsteptol=STOL;

  /* Call KINCreate/KINMalloc to initialize KINSOL:
```

```
     nvSpec is the nvSpec pointer used in the serial version
     A pointer to KINSOL problem memory is returned and stored in kmem. */
  kmem = KINCreate();
  if (check_flag((void *)kmem, "KINCreate", 0)) return(1);
  /* Vector cc passed as template vector. */
  flag = KINMalloc(kmem, func, cc);
  if (check_flag(&flag, "KINMalloc", 1)) return(1);

  flag = KINSetFdata(kmem, data);
  if (check_flag(&flag, "KINSetFdata", 1)) return(1);
  flag = KINSetConstraints(kmem, constraints);
  if (check_flag(&flag, "KINSetConstraints", 1)) return(1);
  flag = KINSetFuncNormTol(kmem, fnormtol);
  if (check_flag(&flag, "KINSetFuncNormTol", 1)) return(1);
  flag = KINSetScaledStepTol(kmem, scsteptol);
  if (check_flag(&flag, "KINSetScaledStepTol", 1)) return(1);

  /* We no longer need the constraints vector since KINSetConstraints
     creates a private copy for KINSOL to use. */
  N_VDestroy_Serial(constraints);

  /* Call KINSpgmr to specify the linear solver KINSPGMR with preconditioner
     routines PrecSetupBD and PrecSolveBD, and the pointer to the user block data. */
  maxl = 15;
  maxlrst = 2;
  flag = KINSpgmr(kmem, maxl);
  if (check_flag(&flag, "KINSpgmr", 1)) return(1);

  flag = KINSpgmrSetMaxRestarts(kmem, maxlrst);
  if (check_flag(&flag, "KINSpgmrSetMaxRestarts", 1)) return(1);
  flag = KINSpgmrSetPreconditioner(kmem,
                                   PrecSetupBD,
                                   PrecSolveBD,
                                   data);
  if (check_flag(&flag, "KINSpgmrSetPreconditioner", 1)) return(1);

  /* Print out the problem size, solution parameters, initial guess. */
  PrintHeader(globalstrategy, maxl, maxlrst, fnormtol, scsteptol);

  /* Call KINSol and print output concentration profile */
  flag = KINSol(kmem,           /* KINSol memory block */
                cc,             /* initial guess on input; solution vector */
                globalstrategy, /* global stragegy choice */
                sc,             /* scaling vector, for the variable cc */
                sc);            /* scaling vector for function values fval */
  if (check_flag(&flag, "KINSol", 1)) return(1);

  printf("\n\n\nComputed equilibrium species concentrations:\n");
  PrintOutput(cc);

  /* Print final statistics and free memory */
  PrintFinalStats(kmem);

  N_VDestroy_Serial(cc);
```

```
  N_VDestroy_Serial(sc);
  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) */
      WebRate(xx, yy, cxy, rxy, f_data);

      for(is = 0; is < NUM_SPECIES; is++) {

        /* Differencing in x direction */
        dcyli = *(cxy+is) - *(cxy - idyl + is) ;
        dcyui = *(cxy + idyu + is) - *(cxy+is);

        /* Differencing in y direction */
        dcxli = *(cxy+is) - *(cxy - idxl + is);
        dcxri = *(cxy + idxr +is) - *(cxy+is);

        /* Compute the total rate value at (xx,yy) */
        fxy[is] = (coy)[is] * (dcyui - dcyli) +
          (cox)[is] * (dcxri - dcxli) + rxy[is];

      } /* end of is loop */

    } /* end of jx loop */

  } /* end of jy loop */
}

/*
 * Preconditioner setup routine. Generate and preprocess P.
 */

static int PrecSetupBD(N_Vector cc, N_Vector cscale,
                       N_Vector fval, N_Vector fscale,
                       void *P_data,
                       N_Vector vtemp1, N_Vector vtemp2)
{
  realtype r, r0, uround, sqruround, xx, yy, delx, dely, csave, fac;
  realtype *cxy, *scxy, **Pxy, *ratesxy, *Pxycol, perturb_rates[NUM_SPECIES];
  long int i, j, jx, jy, ret;
  UserData data;

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

  uround = data->uround;
  sqruround = data->sqruround;
  fac = N_VWL2Norm(fval, fscale);
  r0 = THOUSAND * uround * fac * NEQ;
  if(r0 == ZERO) r0 = ONE;

  /* Loop over spatial points; get size NUM_SPECIES Jacobian block at each */
  for (jy = 0; jy < MY; jy++) {
    yy = jy*dely;

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

```
      xx = jx*delx;
      Pxy = (data->P)[jx][jy];
      cxy = IJ_Vptr(cc,jx,jy);
      scxy= IJ_Vptr(cscale,jx,jy);
      ratesxy = IJ_Vptr((data->rates),jx,jy);

      /* Compute difference quotients of interaction rate fn. */
      for (j = 0; j < NUM_SPECIES; j++) {

        csave = cxy[j];  /* Save the j,jx,jy element of cc */
        r = MAX(sqruround*ABS(csave), r0/scxy[j]);
        cxy[j] += r; /* Perturb the j,jx,jy element of cc */
        fac = ONE/r;

        WebRate(xx, yy, cxy, perturb_rates, data);

        /* Restore j,jx,jy element of cc */
        cxy[j] = csave;

        /* Load the j-th column of difference quotients */
        Pxycol = Pxy[j];
        for (i = 0; i < NUM_SPECIES; i++)
          Pxycol[i] = (perturb_rates[i] - ratesxy[i]) * fac;


      } /* end of j loop */

      /* Do LU decomposition of size NUM_SPECIES preconditioner block */
      ret = gefa(Pxy, NUM_SPECIES, (data->pivot)[jx][jy]);
      if (ret != 0) return(1);

    } /* end of jx loop */

  } /* end of jy loop */

  return(0);
}

/*
 * Preconditioner solve routine
 */

static int PrecSolveBD(N_Vector cc, N_Vector cscale,
                       N_Vector fval, N_Vector fscale,
                       N_Vector vv, void *P_data,
                       N_Vector ftem)
{
  realtype **Pxy, *vxy;
  long int *piv, jx, jy;
  UserData data;

  data = (UserData)P_data;

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

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

      /* For each (jx,jy), solve a linear system of size NUM_SPECIES.
         vxy is the address of the corresponding portion of the vector vv;
         Pxy is the address of the corresponding block of the matrix P;
         piv is the address of the corresponding block of the array pivot. */
      vxy = IJ_Vptr(vv,jx,jy);
      Pxy = (data->P)[jx][jy];
      piv = (data->pivot)[jx][jy];
      gesl (Pxy, NUM_SPECIES, piv, vxy);

    } /* end of jy loop */

  } /* end of jx loop */

  return(0);
}

/*
 * Interaction rate function routine
 */

static void WebRate(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
                    void *f_data)
{
  long int i;
  realtype fac;
  UserData data;

  data = (UserData)f_data;

  for (i = 0; i<NUM_SPECIES; i++)
    ratesxy[i] = DotProd(NUM_SPECIES, cxy, acoef[i]);

  fac = ONE + ALPHA * xx * yy;

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

/*
 * Dot product routine for realtype arrays
 */

static realtype DotProd(long int size, realtype *x1, realtype *x2)
{
  long int i;
  realtype *xx1, *xx2, temp = ZERO;

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

  return(temp);
```

```
}

/*
 *-----------------------------------------------------------------
 * 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);
```

```
}

/*
 * Set initial conditions in cc
 */

static void SetInitialProfiles(N_Vector cc, N_Vector sc)
{
  int i, jx, jy;
  realtype *cloc, *sloc;
  realtype  ctemp[NUM_SPECIES], stemp[NUM_SPECIES];

  /* Initialize arrays ctemp and stemp used in the loading process */
  for (i = 0; i < NUM_SPECIES/2; i++) {
    ctemp[i] = PREYIN;
    stemp[i] = ONE;
  }
  for (i = NUM_SPECIES/2; i < NUM_SPECIES; i++) {
    ctemp[i] = PREDIN;
    stemp[i] = RCONST(0.00001);
  }

  /* Load initial profiles into cc and sc vector from ctemp and stemp. */
  for (jy = 0; jy < MY; jy++) {
    for (jx = 0; jx < MX; jx++) {
      cloc = IJ_Vptr(cc,jx,jy);
      sloc = IJ_Vptr(sc,jx,jy);
      for (i = 0; i < NUM_SPECIES; i++) {
        cloc[i] = ctemp[i];
        sloc[i] = stemp[i];
      }
    }
  }
}

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

static void PrintHeader(int globalstrategy, int maxl, int maxlrst,
                        realtype fnormtol, realtype scsteptol)
{
  printf("\nPredator-prey test problem --  KINSol (serial version)\n\n");
  printf("Mesh dimensions = %d X %d\n", MX, MY);
  printf("Number of species = %d\n", NUM_SPECIES);
  printf("Total system size = %d\n\n", NEQ);
  printf("Flag globalstrategy = %d (1 = Inex. Newton, 2 = Linesearch)\n",
         globalstrategy);
  printf("Linear solver is SPGMR with maxl = %d, maxlrst = %d\n",
         maxl, maxlrst);
  printf("Preconditioning uses interaction-only block-diagonal matrix\n");
  printf("Positivity constraints imposed on all components \n");
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("Tolerance parameters:  fnormtol = %Lg   scsteptol = %Lg\n",
```

```
         fnormtol, scsteptol);
#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++) {
    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
  }
  printf("\n\n");
}

/*
 * Print final statistics contained in iopt
 */

static void PrintFinalStats(void *kmem)
{
  long int nni, nfe, nli, npe, nps, ncfl, nfeSG;
  int flag;

  flag = KINGetNumNonlinSolvIters(kmem, &nni);
  check_flag(&flag, "KINGetNumNonlinSolvIters", 1);
  flag = KINGetNumFuncEvals(kmem, &nfe);
  check_flag(&flag, "KINGetNumFuncEvals", 1);
  flag = KINSpgmrGetNumLinIters(kmem, &nli);
  check_flag(&flag, "KINSpgmrGetNumLinIters", 1);
  flag = KINSpgmrGetNumPrecEvals(kmem, &npe);
  check_flag(&flag, "KINSpgmrGetNumPrecEvals", 1);
  flag = KINSpgmrGetNumPrecSolves(kmem, &nps);
  check_flag(&flag, "KINSpgmrGetNumPrecSolves", 1);
  flag = KINSpgmrGetNumConvFails(kmem, &ncfl);
  check_flag(&flag, "KINSpgmrGetNumConvFails", 1);
  flag = KINSpgmrGetNumFuncEvals(kmem, &nfeSG);
  check_flag(&flag, "KINSpgmrGetNumFuncEvals", 1);

  printf("\nFinal Statistics.. \n\n");
  printf("nni    = %5ld    nli   = %5ld\n", nni, nli);
  printf("nfe    = %5ld    nfeSG = %5ld\n", nfe, nfeSG);
  printf("nps    = %5ld    npe   = %5ld     ncfl  = %5ld\n", nps, npe, ncfl);

}

/*
 * Check function return value...
 *    opt == 0 means SUNDIALS function allocates memory so check if
 *             returned NULL pointer
 *    opt == 1 means SUNDIALS function returns a flag so check if
 *             flag >= 0
 *    opt == 2 means function allocates memory so check if returned
 *             NULL pointer
 */
```

```
static int check_flag(void *flagvalue, char *funcname, int opt)
{
  int *errflag;

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

  /* 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: 1.18.2.2 $
 * $Date: 2005/04/07 00:15:40 $
 * -----------------------------------------------------------------
 * 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 */
```

```
static UserData AllocUserData(void);
static void InitUserData(long int my_pe, long int Nlocal, MPI_Comm comm, UserData data);
static void FreeUserData(UserData data);
static void SetInitialProfiles(N_Vector cc, N_Vector sc);
static void PrintHeader(int globalstrategy, int maxl, int maxlrst,
                        long int mu, long int ml,
                        realtype fnormtol, realtype scsteptol);
static void PrintOutput(long int my_pe, MPI_Comm comm, N_Vector cc);
static void PrintFinalStats(void *kmem);
static void WebRate(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
                    void *f_data);
static realtype DotProd(long int size, realtype *x1, realtype *x2);
static void BSend(MPI_Comm comm, long int my_pe, long int isubx,
                  long int isuby, long int dsizex, long int dsizey,
                  realtype *cdata);
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);
static void BRecvWait(MPI_Request request[], long int isubx,
                      long int isuby, long int dsizex, realtype *cext,
                      realtype *buffer);
static int check_flag(void *flagvalue, char *funcname, int opt, int id);

/*
 *--------------------------------------------------------------------
 * MAIN PROGRAM
 *--------------------------------------------------------------------
 */

int main(int argc, char *argv[])
{
  MPI_Comm comm;
  void *kmem, *pdata;
  UserData data;
  N_Vector cc, sc, constraints;
  int globalstrategy;
  long int Nlocal;
  realtype fnormtol, scsteptol, dq_rel_uu;
  int flag, maxl, maxlrst;
  long int mu, ml;
  int my_pe, npes, npelast = NPEX*NPEY-1;

  data = NULL;
  kmem = pdata = NULL;
  cc = sc = constraints = NULL;

  /* Get processor number and total number of pe's */
  MPI_Init(&argc, &argv);
  comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &my_pe);

```

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

  /* Allocate memory, and set problem data, initial values, tolerances */

  /* Set local length */
  Nlocal = NUM_SPECIES*MXSUB*MYSUB;

  /* Allocate and initialize user data block */
  data = AllocUserData();
  if (check_flag((void *)data, "AllocUserData", 2, my_pe)) MPI_Abort(comm, 1);
  InitUserData(my_pe, Nlocal, comm, data);

  /* Choose global strategy */
  globalstrategy = KIN_INEXACT_NEWTON;

  /* Allocate and initialize vectors */
  cc = N_VNew_Parallel(comm, Nlocal, NEQ);
  if (check_flag((void *)cc, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
  sc = N_VNew_Parallel(comm, Nlocal, NEQ);
  if (check_flag((void *)sc, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
  data->rates = N_VNew_Parallel(comm, Nlocal, NEQ);
  if (check_flag((void *)data->rates, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
  constraints = N_VNew_Parallel(comm, Nlocal, NEQ);
  if (check_flag((void *)constraints, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
  N_VConst(ZERO, constraints);

  SetInitialProfiles(cc, sc);

  fnormtol = FTOL; scsteptol = STOL;

  /* Call KINCreate/KINMalloc to initialize KINSOL:
     nvSpec  points to machine environment data
     A pointer to KINSOL problem memory is returned and stored in kmem. */
  kmem = KINCreate();
  if (check_flag((void *)kmem, "KINCreate", 0, my_pe)) MPI_Abort(comm, 1);

  /* Vector cc passed as template vector. */
  flag = KINMalloc(kmem, func, cc);
  if (check_flag(&flag, "KINMalloc", 1, my_pe)) MPI_Abort(comm, 1);

  flag = KINSetFdata(kmem, data);
  if (check_flag(&flag, "KINSetFdata", 1, my_pe)) MPI_Abort(comm, 1);

  flag = KINSetConstraints(kmem, constraints);
  if (check_flag(&flag, "KINSetConstraints", 1, my_pe)) MPI_Abort(comm, 1);

  /* We no longer need the constraints vector since KINSetConstraints
     creates a private copy for KINSOL to use. */
  N_VDestroy_Parallel(constraints);

```

```
  flag = KINSetFuncNormTol(kmem, fnormtol);
  if (check_flag(&flag, "KINSetFuncNormTol", 1, my_pe)) MPI_Abort(comm, 1);

  flag = KINSetScaledStepTol(kmem, scsteptol);
  if (check_flag(&flag, "KINSetScaledStepTol", 1, my_pe)) MPI_Abort(comm, 1);

  /* Call KINBBDPrecAlloc to initialize and allocate memory for the
     band-block-diagonal preconditioner, and specify the local and
     communication functions func_local and gcomm=NULL (all communication
     needed for the func_local is already done in func). */
  dq_rel_uu = ZERO;
  mu = ml = 2*NUM_SPECIES - 1;

  pdata = KINBBDPrecAlloc(kmem, Nlocal, mu, ml, dq_rel_uu, func_local, NULL);
  if (check_flag((void *)pdata, "KINBBDPrecAlloc", 0, my_pe))
    MPI_Abort(comm, 1);

  /* Call KINBBDSpgmr to specify the linear solver KINSPGMR
     with preconditioner KINBBDPRE */
  maxl = 20; maxlrst = 2;
  flag = KINBBDSpgmr(kmem, maxl, pdata);
  if (check_flag(&flag, "KINBBDSpgmr", 1, my_pe))
    MPI_Abort(comm, 1);

  flag = KINSpgmrSetMaxRestarts(kmem, maxlrst);
  if (check_flag(&flag, "KINSpgmrSetMaxRestarts", 1, my_pe))
    MPI_Abort(comm, 1);

  /* Print out the problem size, solution parameters, initial guess. */
  if (my_pe == 0)
    PrintHeader(globalstrategy, maxl, maxlrst, mu, ml, fnormtol, scsteptol);

  /* call KINSol and print output concentration profile */
  flag = KINSol(kmem,           /* KINSol memory block */
                cc,             /* initial guesss on input; solution vector */
                globalstrategy, /* global stragegy choice */
                sc,             /* scaling vector, for the variable cc */
                sc);            /* scaling vector for function values fval */
  if (check_flag(&flag, "KINSol", 1, my_pe)) MPI_Abort(comm, 1);

  if (my_pe == 0) printf("\n\n\nComputed equilibrium species concentrations:\n");
  if (my_pe == 0 || my_pe==npelast) PrintOutput(my_pe, comm, cc);

  /* Print final statistics and free memory */
  if (my_pe == 0)
    PrintFinalStats(kmem);

  N_VDestroy_Parallel(cc);
  N_VDestroy_Parallel(sc);
  KINBBDPrecFree(pdata);
  KINFree(kmem);
  FreeUserData(data);

  MPI_Finalize();
```

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

/*
 * ccomm routine.  This routine performs all communication
 * between processors of data needed to calculate f.
 */

static void ccomm(long int Nlocal, N_Vector cc, void *userdata)
{

  realtype *cdata, *cext, buffer[2*NUM_SPECIES*MYSUB];
  UserData data;
  MPI_Comm comm;
  long int my_pe, isubx, isuby, nsmxsub, nsmysub;
  MPI_Request request[4];

  /* Get comm, my_pe, subgrid indices, data sizes, extended array cext */
  data = (UserData) userdata;
  comm = data->comm;  my_pe = data->my_pe;
  isubx = data->isubx;   isuby = data->isuby;
  nsmxsub = data->nsmxsub;
  nsmysub = NUM_SPECIES*MYSUB;
  cext = data->cext;

  cdata = NV_DATA_P(cc);

  /* Start receiving boundary data from neighboring PEs */
  BRecvPost(comm, request, my_pe, isubx, isuby, nsmxsub, nsmysub, cext, buffer);

  /* Send data from boundary of local grid to neighboring PEs */
  BSend(comm, my_pe, isubx, isuby, nsmxsub, nsmysub, cdata);

  /* Finish receiving boundary data from neighboring PEs */
  BRecvWait(request, isubx, isuby, nsmxsub, cext, buffer);

}

/*
 * System function for predator-prey system - calculation part
```

```
 */

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;
      for (i = 0; i < NUM_SPECIES; i++) cext[offsetce+i] = cdata[offsetc+i];
    }
  }

  /* Loop over all mesh points, evaluating rate arra at each point */
  delx = data->dx;
  dely = data->dy;
  shifty = (MXSUB+2)*NUM_SPECIES;

  for (jy = 0; jy < MYSUB; jy++) {

    yy = dely*(jy + isuby * MYSUB);

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

      xx = delx * (jx + isubx * MXSUB);
      cxy = IJ_Vptr(cc,jx,jy);
      rxy = IJ_Vptr(data->rates,jx,jy);
      fxy = IJ_Vptr(fval,jx,jy);

      WebRate(xx, yy, cxy, rxy, f_data);

      offsetc = (jx+1)*NUM_SPECIES + (jy+1)*NSMXSUB2;
      offsetcd = offsetc - shifty;
      offsetcu = offsetc + shifty;
      offsetcl = offsetc - NUM_SPECIES;
      offsetcr = offsetc + NUM_SPECIES;

      for (is = 0; is < NUM_SPECIES; is++) {

        /* differencing in x */
        dcydi = cext[offsetc+is]  - cext[offsetcd+is];
        dcyui = cext[offsetcu+is] - cext[offsetc+is];

        /* differencing in y */
        dcxli = cext[offsetc+is]  - cext[offsetcl+is];
        dcxri = cext[offsetcr+is] - cext[offsetc+is];

        /* compute the value at xx , yy */
        fxy[is] = (coy)[is] * (dcyui - dcydi) +
          (cox)[is] * (dcxri - dcxli) + rxy[is];

      } /* end of is loop */

    } /* end of jx loop */

  } /* end of jy loop */
}

/*
 * System function routine.  Evaluate f(cc).  First call ccomm to do
```

```
 * communication of subgrid boundary data into cext.  Then calculate f
 * by a call to func_local.
 */

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

  data = (UserData) f_data;

  /* Call ccomm to do inter-processor communicaiton */
  ccomm(data->Nlocal, cc, data);

  /* Call func_local to calculate all right-hand sides */
  func_local(data->Nlocal, cc, fval, data);
}

/*
 * Interaction rate function routine
 */

static void WebRate(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
                    void *f_data)
{
  long int i;
  realtype fac;
  UserData data;

  data = (UserData)f_data;

  for (i = 0; i<NUM_SPECIES; i++)
    ratesxy[i] = DotProd(NUM_SPECIES, cxy, acoef[i]);

  fac = ONE + ALPHA * xx * yy;

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

/*
 * Dot product routine for realtype arrays
 */

static realtype DotProd(long int size, realtype *x1, realtype *x2)
{
  long int i;
  realtype *xx1, *xx2, temp = ZERO;

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

  return(temp);
}

```

```
/*
 *-----------------------------------------------------------------
 * 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]);
    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)
{

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

  free(data);

}

/*
 * Set initial conditions in cc
 */

static void SetInitialProfiles(N_Vector cc, N_Vector sc)
{
  int i, jx, jy;
  realtype *cloc, *sloc;
  realtype  ctemp[NUM_SPECIES], stemp[NUM_SPECIES];
```

```

  /* Initialize arrays ctemp and stemp used in the loading process */
  for (i = 0; i < NUM_SPECIES/2; i++) {
    ctemp[i] = PREYIN;
    stemp[i] = ONE;
  }
  for (i = NUM_SPECIES/2; i < NUM_SPECIES; i++) {
    ctemp[i] = PREDIN;
    stemp[i] = RCONST(0.00001);
  }

  /* Load initial profiles into cc and sc vector from ctemp and stemp. */
  for (jy = 0; jy < MYSUB; jy++) {
    for (jx=0; jx < MXSUB; jx++) {
      cloc = IJ_Vptr(cc,jx,jy);
      sloc = IJ_Vptr(sc,jx,jy);
      for (i = 0; i < NUM_SPECIES; i++){
        cloc[i] = ctemp[i];
        sloc[i] = stemp[i];
      }
    }
  }

}

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

static void PrintHeader(int globalstrategy, int maxl, int maxlrst,
                        long int mu, long int ml,
                        realtype fnormtol, realtype scsteptol)
{
    printf("\nPredator-prey test problem--  KINSol (parallel-BBD version)\n\n");

    printf("Mesh dimensions = %d X %d\n", MX, MY);
    printf("Number of species = %d\n", NUM_SPECIES);
    printf("Total system size = %d\n\n", NEQ);
    printf("Subgrid dimensions = %d X %d\n", MXSUB, MYSUB);
    printf("Processor array is %d X %d\n\n", NPEX, NPEY);
    printf("Flag globalstrategy = %d (1 = Inex. Newton, 2 = Linesearch)\n",
           globalstrategy);
    printf("Linear solver is SPGMR with maxl = %d, maxlrst = %d\n",
           maxl, maxlrst);
    printf("Preconditioning uses band-block-diagonal matrix from KINBBDPRE\n");
    printf("  with matrix half-bandwidths ml, mu = %ld  %ld\n", ml, mu);
#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf("Tolerance parameters:  fnormtol = %Lg   scsteptol = %Lg\n",
           fnormtol, scsteptol);
#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 sample of current cc values
 */

static void PrintOutput(long int my_pe, MPI_Comm comm, N_Vector cc)
{
  int is, i0, npelast;
  realtype  *ct, tempc[NUM_SPECIES];
  MPI_Status status;

  npelast = NPEX*NPEY - 1;

  ct = NV_DATA_P(cc);

  /* Send the cc values (for all species) at the top right mesh point to PE 0 */
  if (my_pe == npelast) {
    i0 = NUM_SPECIES*(MXSUB*MYSUB-1);
    if (npelast!=0)
      MPI_Send(&ct[i0],NUM_SPECIES,PVEC_REAL_MPI_TYPE,0,0,comm);
    else  /* single processor case */
      for (is = 0; is < NUM_SPECIES; is++) tempc[is]=ct[i0+is];
  }

  /* On PE 0, receive the cc values at top right, then print performance data
     and sampled solution values */
  if (my_pe == 0) {

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

    printf("\nAt bottom left:");
    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
    }

    printf("\n\nAt top right:");
    for (is = 0; is < NUM_SPECIES; is++) {
      if ((is%6)*6 == is) printf("\n");
#if defined(SUNDIALS_EXTENDED_PRECISION)
      printf(" %Lg",tempc[is]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
      printf(" %lg",tempc[is]);
#else
      printf(" %g",tempc[is]);
#endif
    }
    printf("\n\n");
  }
}

/*
 * Print final statistics contained in iopt
 */

static void PrintFinalStats(void *kmem)
{
  long int nni, nfe, nli, npe, nps, ncfl, nfeSG;
  int flag;

  flag = KINGetNumNonlinSolvIters(kmem, &nni);
  check_flag(&flag, "KINGetNumNonlinSolvIters", 1, 0);
  flag = KINGetNumFuncEvals(kmem, &nfe);
  check_flag(&flag, "KINGetNumFuncEvals", 1, 0);
  flag = KINSpgmrGetNumLinIters(kmem, &nli);
  check_flag(&flag, "KINSpgmrGetNumLinIters", 1, 0);
  flag = KINSpgmrGetNumPrecEvals(kmem, &npe);
  check_flag(&flag, "KINSpgmrGetNumPrecEvals", 1, 0);
  flag = KINSpgmrGetNumPrecSolves(kmem, &nps);
  check_flag(&flag, "KINSpgmrGetNumPrecSolves", 1, 0);
  flag = KINSpgmrGetNumConvFails(kmem, &ncfl);
  check_flag(&flag, "KINSpgmrGetNumConvFails", 1, 0);
  flag = KINSpgmrGetNumFuncEvals(kmem, &nfeSG);
  check_flag(&flag, "KINSpgmrGetNumFuncEvals", 1, 0);

  printf("\nFinal Statistics.. \n\n");
  printf("nni    = %5ld    nli   = %5ld\n", nni, nli);
  printf("nfe    = %5ld    nfeSG = %5ld\n", nfe, nfeSG);
  printf("nps    = %5ld    npe   = %5ld     ncfl  = %5ld\n", nps, npe, ncfl);

}

/*
 * Routine to send boundary data to neighboring PEs
 */

```

```c
static void BSend(MPI_Comm comm, long int my_pe,
                  long int isubx, long int isuby,
                  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;

  /* Have bufleft and bufright use the same buffer */
  realtype *bufleft = buffer, *bufright = buffer+NUM_SPECIES*MYSUB;

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

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

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

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

/*
 * Routine to finish 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 BRecvWait(MPI_Request request[], long int isubx,
                      long int isuby, long int dsizex, realtype *cext,
                      realtype *buffer)
{
  int i, ly;
  long int dsizex2, offsetce, offsetbuf;
  realtype *bufleft = buffer, *bufright = buffer+NUM_SPECIES*MYSUB;
  MPI_Status status;

  dsizex2 = dsizex + 2*NUM_SPECIES;

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

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

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

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

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

    /* Copy the buffer to cext */
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NUM_SPECIES;
      offsetce = (ly+2)*dsizex2 - NUM_SPECIES;
      for (i = 0; i < NUM_SPECIES; i++)
        cext[offsetce+i] = bufright[offsetbuf+i];
    }
  }
}
/*
 * Check function return value...
 *    opt == 0 means SUNDIALS function allocates memory so check if
 *             returned NULL pointer
 *    opt == 1 means SUNDIALS function returns a flag so check if
 *             flag >= 0
 *    opt == 2 means function allocates memory so check if returned
 *             NULL pointer
 */

static int check_flag(void *flagvalue, char *funcname, int opt, int id)
{
  int *errflag;

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

  /* Check if flag < 0 */
```

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

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

  return(0);
}
```

C Listing of kindiagsf.f

```
      program kindiagsf
c     ----------------------------------------------------------------
c     $Revision: 1.13.2.1 $
c     $Date: 2005/04/07 00:20:28 $
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 fkinspgmrsetprec(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: 1.13.2.1 $
c     $Date: 2005/04/07 00:20:22 $
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 fkinspgmrsetprec(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
```

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

      subroutine fkfun(uu, fval)

      implicit none

      integer mype, npes
      integer*4 baseadd, nlocal, i, localsize
      parameter(localsize=32)
      double precision pp
      double precision fval(*), uu(*)

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

      do 10 i = 1, nlocal
 10      fval(i) = uu(i) * uu(i) - (i + baseadd) * (i + baseadd)

      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, mype, npes
      integer*4 localsize
      parameter(localsize=32)
      integer*4 baseadd, nlocal, i
      double precision pp
      double precision udata(*), uscale(*), fdata(*), fscale(*)
      double precision vtemp1(*), vtemp2(*)

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

      do 10 i = 1, nlocal
 10      pp(i) = 0.5d0 / (udata(i)+ 5.0d0)

      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, 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
```