# 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

## 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 $8 \times 8$ 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 $6 \times 6$ 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);
```

```
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(void)
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    /* We no longer need the constraints vector since KINSetConstraints
233       creates a private copy for KINSOL to use. */
234    N_VDestroy_Serial(constraints);
235
236    /* Call KINSpgmr to specify the linear solver KINSPGMR with preconditioner
237       routines PrecSetupBD and PrecSolveBD, and the pointer to the user block data. */
238    maxl = 15;
239    maxlrst = 2;
240    flag = KINSpgmr(kmem, maxl);
241    if (check_flag(&flag, "KINSpgmr", 1)) return(1);
242
243    flag = KINSpgmrSetMaxRestarts(kmem, maxlrst);
244    if (check_flag(&flag, "KINSpgmrSetMaxRestarts", 1)) return(1);
245    flag = KINSpgmrSetPreconditioner(kmem,
246                                     PrecSetupBD,
247                                     PrecSolveBD,
248                                     data);
249    if (check_flag(&flag, "KINSpgmrSetPreconditioner", 1)) return(1);
250
251    /* Print out the problem size, solution parameters, initial guess. */
252    PrintHeader(globalstrategy, maxl, maxlrst, fnormtol, scsteptol);
253
254    /* Call KINSol and print output concentration profile */
255    flag = KINSol(kmem,           /* KINSol memory block */
256                  cc,             /* initial guess on input; solution vector */
257                  globalstrategy, /* global stragegy choice */
258                  sc,             /* scaling vector, for the variable cc */
259                  sc);            /* scaling vector for function values fval */
260    if (check_flag(&flag, "KINSol", 1)) return(1);
261
262    printf("\n\n\nComputed equilibrium species concentrations:\n");
263    PrintOutput(cc);
264
265    /* Print final statistics and free memory */
266    PrintFinalStats(kmem);
267
268    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);
```

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

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

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

```
647           fnormtol, scsteptol);
648   #elif defined(SUNDIALS_DOUBLE_PRECISION)
649     printf("Tolerance parameters:  fnormtol = %lg   scsteptol = %lg\n",
650           fnormtol, scsteptol);
651   #else
652     printf("Tolerance parameters:  fnormtol = %g   scsteptol = %g\n",
653           fnormtol, scsteptol);
654   #endif
655
656     printf("\nInitial profile of concentration\n");
657   #if defined(SUNDIALS_EXTENDED_PRECISION)
658     printf("At all mesh points:  %Lg %Lg %Lg   %Lg %Lg %Lg\n",
659           PREYIN, PREYIN, PREYIN,
660           PREDIN, PREDIN, PREDIN);
661   #elif defined(SUNDIALS_DOUBLE_PRECISION)
662     printf("At all mesh points:  %lg %lg %lg   %lg %lg %lg\n",
663           PREYIN, PREYIN, PREYIN,
664           PREDIN, PREDIN, PREDIN);
665   #else
666     printf("At all mesh points:  %g %g %g   %g %g %g\n",
667           PREYIN, PREYIN, PREYIN,
668           PREDIN, PREDIN, PREDIN);
669   #endif
670   }
671
672   /*
673    * Print sampled values of current cc
674    */
675
676   static void PrintOutput(N_Vector cc)
677   {
678     int is, jx, jy;
679     realtype *ct;
680
681     jy = 0; jx = 0;
682     ct = IJ_Vptr(cc,jx,jy);
683     printf("\nAt bottom left:");
684
685     /* Print out lines with up to 6 values per line */
686     for (is = 0; is < NUM_SPECIES; is++){
687       if ((is%6)*6 == is) printf("\n");
688   #if defined(SUNDIALS_EXTENDED_PRECISION)
689       printf(" %Lg",ct[is]);
690   #elif defined(SUNDIALS_DOUBLE_PRECISION)
691       printf(" %lg",ct[is]);
692   #else
693       printf(" %g",ct[is]);
694   #endif
695     }
696
697     jy = MY-1; jx = MX-1;
698     ct = IJ_Vptr(cc,jx,jy);
699     printf("\n\nAt top right:");
700
```

```
  /* 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
 */
```

```
755
756 static int check_flag(void *flagvalue, char *funcname, int opt)
757 {
758   int *errflag;
759
760   /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
761   if (opt == 0 && flagvalue == NULL) {
762     fprintf(stderr,
763             "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
764             funcname);
765     return(1);
766   }
767
768   /* Check if flag < 0 */
769   else if (opt == 1) {
770     errflag = (int *) flagvalue;
771     if (*errflag < 0) {
772       fprintf(stderr,
773               "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
774               funcname, *errflag);
775       return(1);
776     }
777   }
778
779   /* Check if function returned NULL pointer - no memory allocated */
780   else if (opt == 2 && flagvalue == NULL) {
781     fprintf(stderr,
782             "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
783             funcname);
784     return(1);
785   }
786
787   return(0);
788 }
```

# 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 */
```

```
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   /* We no longer need the constraints vector since KINSetConstraints
266      creates a private copy for KINSOL to use. */
267   N_VDestroy_Parallel(constraints);
268
```

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

```
323
324   return(0);
325 }
326
327 /* Readability definitions used in other routines below */
328
329 #define acoef  (data->acoef)
330 #define bcoef  (data->bcoef)
331 #define cox    (data->cox)
332 #define coy    (data->coy)
333
334 /*
335  *--------------------------------------------------------------------
336  * FUNCTIONS CALLED BY KINSOL
337  *--------------------------------------------------------------------
338  */
339
340 /*
341  * ccomm routine.  This routine performs all communication
342  * between processors of data needed to calculate f.
343  */
344
345 static void ccomm(long int Nlocal, N_Vector cc, void *userdata)
346 {
347
348   realtype *cdata, *cext, buffer[2*NUM_SPECIES*MYSUB];
349   UserData data;
350   MPI_Comm comm;
351   long int my_pe, isubx, isuby, nsmxsub, nsmysub;
352   MPI_Request request[4];
353
354   /* Get comm, my_pe, subgrid indices, data sizes, extended array cext */
355   data = (UserData) userdata;
356   comm = data->comm;  my_pe = data->my_pe;
357   isubx = data->isubx;   isuby = data->isuby;
358   nsmxsub = data->nsmxsub;
359   nsmysub = NUM_SPECIES*MYSUB;
360   cext = data->cext;
361
362   cdata = NV_DATA_P(cc);
363
364   /* Start receiving boundary data from neighboring PEs */
365   BRecvPost(comm, request, my_pe, isubx, isuby, nsmxsub, nsmysub, cext, buffer);
366
367   /* Send data from boundary of local grid to neighboring PEs */
368   BSend(comm, my_pe, isubx, isuby, nsmxsub, nsmysub, cdata);
369
370   /* Finish receiving boundary data from neighboring PEs */
371   BRecvWait(request, isubx, isuby, nsmxsub, cext, buffer);
372
373 }
374
375 /*
376  * 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) {
```

```
431       for (ly = 0; ly < MYSUB; ly++) {
432         offsetc = (ly+1)*nsmxsub - 2*NUM_SPECIES;
433         offsetce = (ly+2)*nsmxsub2 - NUM_SPECIES;
434         for (i = 0; i < NUM_SPECIES; i++) cext[offsetce+i] = cdata[offsetc+i];
435       }
436     }
437
438     /* Loop over all mesh points, evaluating rate arra at each point */
439     delx = data->dx;
440     dely = data->dy;
441     shifty = (MXSUB+2)*NUM_SPECIES;
442
443     for (jy = 0; jy < MYSUB; jy++) {
444
445       yy = dely*(jy + isuby * MYSUB);
446
447       for (jx = 0; jx < MXSUB; jx++) {
448
449         xx = delx * (jx + isubx * MXSUB);
450         cxy = IJ_Vptr(cc,jx,jy);
451         rxy = IJ_Vptr(data->rates,jx,jy);
452         fxy = IJ_Vptr(fval,jx,jy);
453
454         WebRate(xx, yy, cxy, rxy, f_data);
455
456         offsetc = (jx+1)*NUM_SPECIES + (jy+1)*NSMXSUB2;
457         offsetcd = offsetc - shifty;
458         offsetcu = offsetc + shifty;
459         offsetcl = offsetc - NUM_SPECIES;
460         offsetcr = offsetc + NUM_SPECIES;
461
462         for (is = 0; is < NUM_SPECIES; is++) {
463
464           /* differencing in x */
465           dcydi = cext[offsetc+is]  - cext[offsetcd+is];
466           dcyui = cext[offsetcu+is] - cext[offsetc+is];
467
468           /* differencing in y */
469           dcxli = cext[offsetc+is]  - cext[offsetcl+is];
470           dcxri = cext[offsetcr+is] - cext[offsetc+is];
471
472           /* compute the value at xx , yy */
473           fxy[is] = (coy)[is] * (dcyui - dcydi) +
474             (cox)[is] * (dcxri - dcxli) + rxy[is];
475
476         } /* end of is loop */
477
478       } /* end of jx loop */
479
480     } /* end of jy loop */
481   }
482
483   /*
484    * System function routine.  Evaluate f(cc).  First call ccomm to do
```

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

```

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

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

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

```

```
809 static void BSend(MPI_Comm comm, long int my_pe,
810                   long int isubx, long int isuby,
811                   long int dsizex, long int dsizey, realtype *cdata)
812 {
813   int i, ly;
814   long int offsetc, offsetbuf;
815   realtype bufleft[NUM_SPECIES*MYSUB], bufright[NUM_SPECIES*MYSUB];
816 
817   /* If isuby > 0, send data from bottom x-line of u */
818   if (isuby != 0)
819     MPI_Send(&cdata[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);
820 
821   /* If isuby < NPEY-1, send data from top x-line of u */
822   if (isuby != NPEY-1) {
823     offsetc = (MYSUB-1)*dsizex;
824     MPI_Send(&cdata[offsetc], dsizex, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
825   }
826 
827   /* If isubx > 0, send data from left y-line of u (via bufleft) */
828   if (isubx != 0) {
829     for (ly = 0; ly < MYSUB; ly++) {
830       offsetbuf = ly*NUM_SPECIES;
831       offsetc = ly*dsizex;
832       for (i = 0; i < NUM_SPECIES; i++)
833         bufleft[offsetbuf+i] = cdata[offsetc+i];
834     }
835     MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
836   }
837 
838   /* If isubx < NPEX-1, send data from right y-line of u (via bufright) */
839   if (isubx != NPEX-1) {
840     for (ly = 0; ly < MYSUB; ly++) {
841       offsetbuf = ly*NUM_SPECIES;
842       offsetc = offsetbuf*MXSUB + (MXSUB-1)*NUM_SPECIES;
843       for (i = 0; i < NUM_SPECIES; i++)
844         bufright[offsetbuf+i] = cdata[offsetc+i];
845     }
846     MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
847   }
848 }
849 
850 /*
851  * Routine to start receiving boundary data from neighboring PEs.
852  *  Notes:
853  *  1) buffer should be able to hold 2*NUM_SPECIES*MYSUB realtype entries,
854  *     should be passed to both the BRecvPost and BRecvWait functions, and
855  *     should not be manipulated between the two calls.
856  *  2) request should have 4 entries, and should be passed in both calls also.
857  */
858 
859 static void BRecvPost(MPI_Comm comm, MPI_Request request[], long int my_pe,
860                       long int isubx, long int isuby,
861                       long int dsizex, long int dsizey,
862                       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);
```

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

```
971   else if (opt == 1) {
972     errflag = (int *) flagvalue;
973     if (*errflag < 0) {
974       fprintf(stderr,
975               "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
976               id, funcname, *errflag);
977       return(1);
978     }
979   }
980
981   /* Check if function returned NULL pointer - no memory allocated */
982   else if (opt == 2 && flagvalue == NULL) {
983     fprintf(stderr,
984             "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
985             id, funcname);
986     return(1);
987   }
988
989   return(0);
990 }
```

# 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
```

```
53        endif
54
55        do 20 i = 1, neq
56           uu(i) = 2.0d0 * i
57           scale(i) = 1.0d0
58           constr(i) = 0.0d0
59     20 continue
60
61        call fkinmalloc(msbpre, fnormtol, scsteptol,
62       &                constr, inopt, iopt, ropt, ier)
63        if (ier .ne. 0) then
64           write(6,1230) ier
65     1230    format('SUNDIALS_ERROR: FKINMALLOC returned IER = ', i2)
66           call fnvfrees
67           stop
68        endif
69
70        call fkinspgmr(maxl, maxlrst, ier)
71        if (ier .ne. 0) then
72           write(6,1235) ier
73     1235    format('SUNDIALS_ERROR: FKINSPGMR returned IER = ', i2)
74           call fkinfree
75           call fnvfrees
76           stop
77        endif
78
79        call fkinspgmrsetprec(1, ier)
80
81        write(6,1240)
82     1240 format('Example program kindiagsf:'//' This fkinsol example code',
83       1       ' solves a 128 eqn diagonal algebraic system.'/
84       2       ' Its purpose is to demonstrate the use of the Fortran',
85       3       ' interface'/' in a serial environment.'///
86       4       ' globalstrategy = KIN_INEXACT_NEWTON')
87
88        call fkinsol(uu, globalstrat, scale, scale, ier)
89        if (ier .lt. 0) then
90           write(6,1242) ier, iopt(15)
91     1242    format('SUNDIALS_ERROR: FKINSOL returned IER = ', i2, /,
92       1          '                Linear Solver returned IER = ', i2)
93           call fkinfree
94           call fnvfrees
95           stop
96        endif
97
98        write(6,1245) ier
99     1245 format(/' FKINSOL return code is ', i3)
100
101       write(6,1246)
102    1246 format(//' The resultant values of uu are:'/)
103
104       do 30 i = 1, neq, 4
105          write(6,1256) i, uu(i), uu(i+1), uu(i+2), uu(i+3)
106    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`

```
1         program kindiagpf
2   c     ----------------------------------------------------------------
3   c     $Revision: 1.13.2.1 $
4   c     $Date: 2005/04/07 00:20:22 $
5   c     ----------------------------------------------------------------
6   c     Programmer(s): Allan G. Taylor, Alan C. Hindmarsh and
7   c                    Radu Serban @ LLNL
8   c     ----------------------------------------------------------------
9   c     Simple diagonal test with Fortran interface, using
10  c     user-supplied preconditioner setup and solve routines (supplied
11  c     in Fortran, below).
12  c
13  c     This example does a basic test of the solver by solving the
14  c     system:
15  c                f(u) = 0  for
16  c                f(u) = u(i)^2 - i^2
17  c
18  c      No scaling is done.
19  c      An approximate diagonal preconditioner is used.
20  c
21  c      Execution command: mpirun -np 4 kindiagpf
22  c     ----------------------------------------------------------------
23  c
24        include "mpif.h"
25
26        integer ier, size, globalstrat, rank, inopt, mype, npes
27        integer maxl, maxlrst
28        integer*4 localsize
29        parameter(localsize=32)
30        integer*4 neq, nlocal, msbpre, baseadd, i, ii
31        integer*4 iopt(40)
32        double precision pp, fnormtol, scsteptol
33        double precision uu(localsize), scale(localsize)
34        double precision constr(localsize)
35
36        common /pcom/ pp(localsize), mype, npes, baseadd, nlocal
37
38        nlocal = localsize
39        neq = 4 * nlocal
40        globalstrat = 1
41        fnormtol = 1.0d-5
42        scsteptol = 1.0d-4
43        inopt = 0
44        maxl = 10
45        maxlrst = 2
46        msbpre  = 5
47
48  c     The user MUST call mpi_init, Fortran binding, for the fkinsol package
49  c     to work. The communicator, MPI_COMM_WORLD, is the only one common
50  c     between the Fortran and C bindings. So in the following, the communicator
51  c     MPI_COMM_WORLD is used in calls to mpi_comm_size and mpi_comm_rank
52  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
```

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