Example Programs for KINSOL v2.5.0

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

November 6, 2006

UCRL-SM-208114

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

Contents

1 Introduction

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

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

Supplied in the `sundials-x.y.z/examples/kinsol/serial` directory are the following serial examples (using the NVECTOR_SERIAL module):

- `kinbanx` solves a simple 2-D elliptic PDE on a unit square.
  This program solves the problem with the KINBAND linear solver.

- `kindenx1` solves the Ferraris-Tronconi problem.
  This program solves the problem with the KINDENSE linear solver and uses different combinations of globalization and Jacobian update strategies with different initial guesses.

- `kindenx2` solves a nonlinear system from robot kinematics.
  This program solves the problem with the KINDENSE linear solver and a user-supplied Jacobian routine.

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

- `kinkrydem_lin` solves the same problem as `kinkryx`, but with three Krylov linear solvers: `kinspgmr`, `kinspbcg`, and `kinsptfqmr`.

Supplied in the `sundials-x.y.z/examples/kinsol/parallel` directory are the following parallel examples (using the NVECTOR_PARALLEL module):

- `kinkryx_p` is a parallel implementation of `kinkryx`.

- `kinkryx_bbd_p` solves the same problem as `kinkryx_p`, with a block-diagonal matrix with banded blocks as a preconditioner, generated by difference quotients, using the KINBBDPRE module.

With the FKINSOL module, in the directories `sundials-x.y.z/examples/kinsol/fcmix_serial` and `sundials-x.y.z/examples/kinsol/fcmix_parallel`, are the following examples for the FORTRAN-C interface:

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

- `fkinkryx_p` is a parallel implementation of `fkinkryx`.

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. §5.2) 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 used 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 dense example: kindenx1

As an initial illustration of the use of the KINSOL package for the solution of nonlinear systems, we give a sample program called `kindenx1.c`. It uses the KINSOL dense linear solver module KINDENSE and the NVECTOR_SERIAL module (which provides a serial implementation of NVECTOR) for the solution of the Ferraris-Tronconi test problem [2].

This problem involves a blend of trigonometric and exponential terms:

$$\begin{aligned} 0 &= 0.5\sin(x_1 x_2) - 0.25 x_2/\pi - 0.5 x_1 \\ 0 &= (1 - 0.25/\pi)(e^{2x_1} - e) + e x_2/\pi - 2e x_1 \\ &\text{subject to} \\ &\quad x_{1\,\min} = 0.25 \le x_1 \le 1 = x_{1\,\max} \\ &\quad x_{2\,\min} = 1.5 \le x_2 \le 2\pi = x_{2\,\max}\,. \end{aligned} \tag{1}$$

The bounds constraints on x_1 and x_2 are treated by introducing 4 additional variables and using KINSOL's optional constraints feature to enforce non-positivity and non-negativity:

$$\begin{aligned} l_1 &= x_1 - x_{1\,\min} \ge 0 \\ L_1 &= x_1 - x_{1\,\max} \le 0 \\ l_2 &= x_2 - x_{2\,\min} \ge 0 \\ L_2 &= x_2 - x_{2\,\max} \le 0\,. \end{aligned}$$

The Ferraris-Tronconi problem has two known solutions. We solve it with KINSOL using two sets of initial guesses for x_1 and x_2 (first their lower bounds and secondly the middle of their feasible regions), both with an exact and modified Newton method, with and without line search. The source code is listed in Appendix A.

Following the initial comment block, this program has a number of `#include` lines, which allow access to useful items in CVODE header files. The `kinsol.h` file provides prototypes for the KINSOL functions to be called (excluding the linear solver selection function), and also a number of constants that are to be used in setting input arguments and testing the return value of `KINSol`. The `nvector_serial.h` file is the header file for the serial implementation of the NVECTOR module and includes definitions of the `N_Vector` type, a macro to access vector components, and prototypes for the serial implementation specific machine environment memory allocation and freeing functions. The `kinsol_dense.h` file provides the prototype for the `KINDense` function. The `sundials_types.h` file provides the definition of the type `realtype` (see §5.2 for details). For now, it suffices to read `realtype` as `double`. Finally, `sundials_math.h` is included for the definition of the exponential functin `RExp`.

Next, the program defines some problem-specific constants, which are isolated to this early location to make it easy to change them as needed. This program includes a user-defined accessor macros, `Ith` that is useful in writing the problem functions in a form closely matching the mathematical description of the system, i.e. with components numbered from 1 instead of from 0. The `Ith` macro is used to access components of a vector of type `N_Vector` with a serial implementation. It is defined using the NVECTOR_SERIAL accessor macro `NV_Ith_S` which numbers components starting with 0. The program prologue ends with prototypes of the user-supplied system function `func` and several private helper functions.

The `main` program begins with some dimensions and type declarations, including use of the type `N_Vector`, initializations, and allocation and definitions for the user data structure

data which contains two arrays with lower and upper bounds for x_1 and x_2. Next, we create 5 serial vectors of type N_Vector for the two different initial guesses, the solution vector u, the scaling factors, and the constraint specifications.

The initial guess vectors u1 and u2 are initialized in the private functions SetInitialGuess1 and SetInitialGuess2 and the constraint vector c is initialized to $[0, 0, 1, -1, 1, -1]$ indicating that there are no additional constraints on the first two components of u (i.e.; x_1 and x_2) and that the 3rd and 5th compnents should be non-negative, while the 4th and 6th should be non-positive.

The call to KINCreate creates the KINSOL solver memory block. Its return value is a pointer to that memory block for this problem. In the case of failure, the return value is NULL. This pointer must be passed in the remaining calls to KINSOL functions.

The next 4 calls to KINSOL optional input functions specify the pointer to the user data structure (to be passed to all subsequent calls to func), the vector of additional constraints, and the function and scaled step tolerances, fnormtol and scsteptol, respectively.

Solver memory is allocated through the call to KINMalloc which specifies the system function func and provides the vector u which will be used internally as a template for clonning additional necessary vectors of the same type as u. The use of the dense linear solver is specifid by calling KINDense which also specifies the problem size NEQ.

The main program proceeds by solving the nonlinear system 6 times, using each of the two initial guesses u1 and u2 (which are first copied into the vector u using the N_VScale_Serial function from the NVECTOR_SERIAL module), with and without globalization through line search (specified by setting glstr to KIN_LINESEARCH and KIN_NONE, respectively), and applying either an exact or a modified Newton method. The switch from exact to modified Newton is done by changing the number of nonlinear iterations after which a Jacobian evaluation is enforced, a value mset$= 1$ thus resulting in re-evaluating the Jacobian at every single iteratin of the nonlinear solver (exact Newton method). Note that passing mset$= 0$ indicates using the default KINSOL value of 10.

The actual problem solution is carried out in the private function SolveIt which calls the main solver function KINSol after first setting the optional input mset. After a successful return from KINSol, the solution $[x_1, x_2]$ and some solver statistics are printed.

The function func is a straightforward expression of the extended nonlinear system. It uses the macro NV_DATA_S (defined by the NVECTOR_SERIAL module) to extract the pointers to the data arrays of the N_Vectors u and f and sets the components of fdata using the current values for the components of udata. See §5.6.1 for a detailed specification of f.

The output generated by kindenx1 is shown below.

kindenx1 sample output

```
Ferraris and Tronconi test problem
Tolerance parameters:
  fnormtol  =      1e-05
  scsteptol =      1e-05

------------------------------------------

Initial guess on lower bounds
  [x1,x2] =      0.25       1.5

Exact Newton
Solution:
```

```
  [x1,x2] =  0.299449   2.83693
Final Statistics:
  nni =     3    nfe  =     4
  nje =     3    nfeD =    18

Exact Newton with line search
Solution:
  [x1,x2] =  0.299449   2.83693
Final Statistics:
  nni =     3    nfe  =     4
  nje =     3    nfeD =    18

Modified Newton
Solution:
  [x1,x2] =  0.299449   2.83693
Final Statistics:
  nni =    11    nfe  =    12
  nje =     2    nfeD =    12

Modified Newton with line search
Solution:
  [x1,x2] =  0.299449   2.83693
Final Statistics:
  nni =    11    nfe  =    12
  nje =     2    nfeD =    12

----------------------------------------

Initial guess in middle of feasible region
  [x1,x2] =     0.625   3.89159

Exact Newton
Solution:
  [x1,x2] =       0.5   3.14159
Final Statistics:
  nni =     5    nfe  =     6
  nje =     5    nfeD =    30

Exact Newton with line search
Solution:
  [x1,x2] =       0.5   3.14159
Final Statistics:
  nni =     5    nfe  =     6
  nje =     5    nfeD =    30

Modified Newton
Solution:
  [x1,x2] =  0.500003    3.1416
Final Statistics:
  nni =    12    nfe  =    13
  nje =     2    nfeD =    12

Modified Newton with line search
Solution:
  [x1,x2] =  0.500003    3.1416
Final Statistics:
  nni =    12    nfe  =    13
  nje =     2    nfeD =    12
```

2.2 A serial Krylov example: kinkryx

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, `kinkryx.c`, is listed in Appendix B.

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

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

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

$$f_i(x, y, c) = c_i \cdot \left(b_i + \sum_{j=1}^{n_s} a_{i,j} \cdot c_j \right)\,. \tag{3}$$

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 (2) are discretized spatially with standard central finite differences on a 8×8 mesh with $n_s = 6$, giving a system of size 384.

Among the initial `#include` lines in this case are lines to include `kinso_spgmr.h` and `sundials_math.h`. The first contains constants and function prototypes associated with the SPGMR method. The inclusion of `sundials_math.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 §5.5.2) 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 KINSpilsSetPreconditioner (see §5.5.4). The data pointer passed to KINSpilsSetPreconditioner 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 §5.5.5.

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 (3) 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 §5.6.8 for detailed descriptions of these preconditioner functions.

The program kinkryx.c uses the "small" dense functions for all operations on the 6×6 preconditioner blocks. Thus it includes sundials_smalldense.h, and calls the small dense matrix functions denalloc, denallocpiv, denfree, denfreepiv, denGETRF, and denGETRS. The small dense functions are generally available for KINSOL user programs (for more information, see §9.1 or the comments in the header file sundials_smalldense.h).

In addition to the functions called by KINSOL, kinkryx.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 kinkryx is shown below. Note that the solution involved 7 Newton iterations, with an average of about 33 Krylov iterations per Newton iteration.

kinkryx sample output

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

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

Flag globalstrategy = 0 (0 = None, 1 = 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    =    10    nli   =   378
nfe    =    11    nfeSG =   388
nps    =   388    npe   =     1     ncfl  =     7
```

2.3 A parallel example: kinkryx_bbd_p

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

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 nonlinears 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.

KINBBDPRE uses a band-block-diagonal preconditioner, generated by difference quotients. The upper and lower half-bandwidths of the Jacobian block generated on each process are both equal to $2 \cdot n_s - 1$, and that is the value passeded as `mudq` and `mldq` in the call to `KINBBDPrecAlloc`. These values are much less than the true half-bandwidths of the Jacobian blocks, which are $n_s \cdot$ `MXSUB`. However, an even narrower band matrix is retained as the preconditioner, with half-bandwidths equal to n_s, and this is the value passed to `KINBBDPrecAlloc` for `mu` and `ml`.

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 `kinkryx` 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 rank 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 `kinkryx_bbd_p` is shown below. Note that 9 Newton iterations were required, with an average of about 51.6 Krylov iterations per Newton iteration.

kinkryx_bbd_p 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 = 0 (0 = None, 1 = Linesearch)
Linear solver is SPGMR with maxl = 20, maxlrst = 2
Preconditioning uses band-block-diagonal matrix from KINBBDPRE
  Difference quotient half-bandwidths are mudq = 11, mldq = 11
  Retained band block half-bandwidths are mukeep = 6, mlkeep = 6
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    =     9    nli   =   464
nfe    =    10    nfeSG =   473
nps    =   473    npe   =     1     ncfl  =     6
```

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: fkinkryx

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

The main program begins by calling `fnvinits` to initialize computations with the 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 `iout` and `rout` arrays which are used to store integer and real outputs, respectively (see Table 6.2). Also, various integer, real and vector parameters are specified by calling the `fkinsetiin`, `fkinsetrin`, and `fkinsetvin` subroutines, respectively. In particular, 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}), and the tolerance for stopping based on the step length (`scsteptol` = 10^{-4}) are specified.

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 `fkinspilssetprec` 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).

Finally, memory allocated for the KINSOL solver is released by calling `fkinfree`.

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 `fkinkryx`, using $N = 128$.

fkinkryx sample output

```
Example program fkinkryx:
```

```
 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_NONE

 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: fkinkryx_p

The program `kindiapf`, listed in Appendix E, is a straightforward modification of `fkinkryx` to use the MPI-enabled NVECTOR_PARALLEL module.

After initialization of MPI, the NVECTOR_PARALLEL module is initialized by calling

fnvinitp with the default MPI communicator mpi_comm_world and local and global vector sizes as its first three arguments. The problem set-up (KINSOL initialization, KINSPGMR specification) and solution steps are the same as in fkinkryx. 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 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 fkinkryx.

Sample output from fkinkryx_p, for $N = 128$, follows.

fkinkryx_p sample output

```
Example program fkinkryx_p:

 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.

 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 v2.4.0. Technical Report UCRL-SM-208116, LLNL, 2006.

[2] C. Floudas, P. Pardalos, C. Adjiman, W. Esposito, Z. Gumus, S. Harding, J. Klepeis, C. Meyer, and C. Schweiger. *Handbook of Test Problems in Local and Global Optimization.* Kluwer Academic Publishers, Dordrecht, 1999.

A Listing of kindenx1.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.1 $
 * $Date: 2006/07/05 15:50:11 $
 * -----------------------------------------------------------------
 * Programmer(s): Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example (serial):
 *
 * This example solves a nonlinear system from.
 *
 * Source: "Handbook of Test Problems in Local and Global Optimization",
 *             C.A. Floudas, P.M. Pardalos et al.
 *             Kluwer Academic Publishers, 1999.
 * Test problem 4 from Section 14.1, Chapter 14: Ferraris and Tronconi
 *
 * This problem involves a blend of trigonometric and exponential terms.
 *    0.5 sin(x1 x2) - 0.25 x2/pi - 0.5 x1 = 0
 *    (1-0.25/pi) ( exp(2 x1)-e ) + e x2 / pi - 2 e x1 = 0
 * such that
 *    0.25 <= x1 <=1.0
 *    1.5 <= x2 <= 2 pi
 *
 * The treatment of the bound constraints on x1 and x2 is done using
 * the additional variables
 *    l1 = x1 - x1_min >= 0
 *    L1 = x1 - x1_max <= 0
 *    l2 = x2 - x2_min >= 0
 *    L2 = x2 - x2_max >= 0
 *
 * and using the constraint feature in KINSOL to impose
 *    l1 >= 0    l2 >= 0
 *    L1 <= 0    L2 <= 0
 *
 * The Ferraris-Tronconi test problem has two known solutions.
 * The nonlinear system is solved by KINSOL using different
 * combinations of globalization and Jacobian update strategies
 * and with different initial guesses (leading to one or the other
 * of the known solutions).
 *
 *
 * Constraints are imposed to make all components of the solution
 * positive.
 * -----------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <kinsol/kinsol.h>
#include <kinsol/kinsol_dense.h>
#include <nvector/nvector_serial.h>
#include <sundials/sundials_types.h>
#include <sundials/sundials_math.h>

/* Problem Constants */
```

```c

#define NVAR   2
#define NEQ    3*NVAR

#define FTOL   RCONST(1.e-5)  /* function tolerance */
#define STOL   RCONST(1.e-5) /* step tolerance */

#define ZERO   RCONST(0.0)
#define PT25   RCONST(0.25)
#define PT5    RCONST(0.5)
#define ONE    RCONST(1.0)
#define ONEPT5 RCONST(1.5)
#define TWO    RCONST(2.0)

#define PI     RCONST(3.1415926)
#define E      RCONST(2.7182818)

typedef struct {
  realtype lb[NVAR];
  realtype ub[NVAR];
} *UserData;

/* Accessor macro */
#define Ith(v,i)    NV_Ith_S(v,i-1)

/* Functions Called by the KINSOL Solver */
static int func(N_Vector u, N_Vector f, void *f_data);

/* Private Helper Functions */
static void SetInitialGuess1(N_Vector u, UserData data);
static void SetInitialGuess2(N_Vector u, UserData data);
static int SolveIt(void *kmem, N_Vector u, N_Vector s, int glstr, int mset);
static void PrintHeader(int globalstrategy, realtype fnormtol, realtype scsteptol);
static void PrintOutput(N_Vector u);
static void PrintFinalStats(void *kmem);
static int check_flag(void *flagvalue, char *funcname, int opt);

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

int main()
{
  UserData data;
  realtype fnormtol, scsteptol;
  N_Vector u1, u2, u, s, c;
  int glstr, mset, flag;
  void *kmem;

  u1 = u2 = u = NULL;
  s = c = NULL;
  kmem = NULL;
  data = NULL;
  glstr = KIN_NONE;

  /* User data */

```

```
117     data = (UserData)malloc(sizeof *data);
118     data->lb[0] = PT25;       data->ub[0] = ONE;
119     data->lb[1] = ONEPT5;     data->ub[1] = TWO*PI;
120
121     /* Create serial vectors of length NEQ */
122     u1 = N_VNew_Serial(NEQ);
123     if (check_flag((void *)u1, "N_VNew_Serial", 0)) return(1);
124
125     u2 = N_VNew_Serial(NEQ);
126     if (check_flag((void *)u2, "N_VNew_Serial", 0)) return(1);
127
128     u = N_VNew_Serial(NEQ);
129     if (check_flag((void *)u, "N_VNew_Serial", 0)) return(1);
130
131     s = N_VNew_Serial(NEQ);
132     if (check_flag((void *)s, "N_VNew_Serial", 0)) return(1);
133
134     c = N_VNew_Serial(NEQ);
135     if (check_flag((void *)c, "N_VNew_Serial", 0)) return(1);
136
137     SetInitialGuess1(u1,data);
138     SetInitialGuess2(u2,data);
139
140     N_VConst_Serial(ONE,s); /* no scaling */
141
142     Ith(c,1) =  ZERO;   /* no constraint on x1 */
143     Ith(c,2) =  ZERO;   /* no constraint on x2 */
144     Ith(c,3) =  ONE;    /* l1 = x1 - x1_min >= 0 */
145     Ith(c,4) = -ONE;    /* L1 = x1 - x1_max <= 0 */
146     Ith(c,5) =  ONE;    /* l2 = x2 - x2_min >= 0 */
147     Ith(c,6) = -ONE;    /* L2 = x2 - x22_min <= 0 */
148
149     fnormtol=FTOL; scsteptol=STOL;
150
151
152     kmem = KINCreate();
153     if (check_flag((void *)kmem, "KINCreate", 0)) return(1);
154
155     flag = KINSetFdata(kmem, data);
156     if (check_flag(&flag, "KINSetFdata", 1)) return(1);
157     flag = KINSetConstraints(kmem, c);
158     if (check_flag(&flag, "KINSetConstraints", 1)) return(1);
159     flag = KINSetFuncNormTol(kmem, fnormtol);
160     if (check_flag(&flag, "KINSetFuncNormTol", 1)) return(1);
161     flag = KINSetScaledStepTol(kmem, scsteptol);
162     if (check_flag(&flag, "KINSetScaledStepTol", 1)) return(1);
163
164     flag = KINMalloc(kmem, func, u);
165     if (check_flag(&flag, "KINMalloc", 1)) return(1);
166
167     /* Call KINDense to specify the linear solver */
168
169     flag = KINDense(kmem, NEQ);
170     if (check_flag(&flag, "KINDense", 1)) return(1);
171
172     /* Print out the problem size, solution parameters, initial guess. */
173     PrintHeader(glstr, fnormtol, scsteptol);
174
175     /* --------------------------- */
```

```
176
177     printf("\n----------------------------------------\n");
178     printf("\nInitial␣guess␣on␣lower␣bounds\n");
179     printf("␣␣[x1,x2]␣=␣");
180     PrintOutput(u1);
181
182     N_VScale_Serial(ONE,u1,u);
183     glstr = KIN_NONE;
184     mset = 1;
185     SolveIt(kmem, u, s, glstr, mset);
186
187     /* --------------------------- */
188
189     N_VScale_Serial(ONE,u1,u);
190     glstr = KIN_LINESEARCH;
191     mset = 1;
192     SolveIt(kmem, u, s, glstr, mset);
193
194     /* --------------------------- */
195
196     N_VScale_Serial(ONE,u1,u);
197     glstr = KIN_NONE;
198     mset = 0;
199     SolveIt(kmem, u, s, glstr, mset);
200
201     /* --------------------------- */
202
203     N_VScale_Serial(ONE,u1,u);
204     glstr = KIN_LINESEARCH;
205     mset = 0;
206     SolveIt(kmem, u, s, glstr, mset);
207
208
209
210     /* --------------------------- */
211
212     printf("\n----------------------------------------\n");
213     printf("\nInitial␣guess␣in␣middle␣of␣feasible␣region\n");
214     printf("␣␣[x1,x2]␣=␣");
215     PrintOutput(u2);
216
217     N_VScale_Serial(ONE,u2,u);
218     glstr = KIN_NONE;
219     mset = 1;
220     SolveIt(kmem, u, s, glstr, mset);
221
222     /* --------------------------- */
223
224     N_VScale_Serial(ONE,u2,u);
225     glstr = KIN_LINESEARCH;
226     mset = 1;
227     SolveIt(kmem, u, s, glstr, mset);
228
229     /* --------------------------- */
230
231     N_VScale_Serial(ONE,u2,u);
232     glstr = KIN_NONE;
233     mset = 0;
234     SolveIt(kmem, u, s, glstr, mset);
```

```
235
236     /* --------------------------- */
237
238     N_VScale_Serial(ONE,u2,u);
239     glstr = KIN_LINESEARCH;
240     mset = 0;
241     SolveIt(kmem, u, s, glstr, mset);
242
243
244
245
246     /* Free memory */
247
248     N_VDestroy_Serial(u);
249     N_VDestroy_Serial(s);
250     N_VDestroy_Serial(c);
251     KINFree(&kmem);
252     free(data);
253
254     return(0);
255   }
256
257
258   static int SolveIt(void *kmem, N_Vector u, N_Vector s, int glstr, int mset)
259   {
260     int flag;
261
262     printf("\n");
263
264     if (mset==1)
265       printf("Exact␣Newton");
266     else
267       printf("Modified␣Newton");
268
269     if (glstr == KIN_NONE)
270       printf("\n");
271     else
272       printf("␣with␣line␣search\n");
273
274     flag = KINSetMaxSetupCalls(kmem, mset);
275     if (check_flag(&flag, "KINSetMaxSetupCalls", 1)) return(1);
276
277     flag = KINSol(kmem, u, glstr, s, s);
278     if (check_flag(&flag, "KINSol", 1)) return(1);
279
280     printf("Solution:\n␣␣[x1,x2]␣=␣");
281     PrintOutput(u);
282
283     PrintFinalStats(kmem);
284
285     return(0);
286
287   }
288
289
290   /*
291    *------------------------------------------------------------------------
292    * FUNCTIONS CALLED BY KINSOL
293    *------------------------------------------------------------------------
```

```
294   */
295
296  /*
297   * System function for predator-prey system
298   */
299
300  static int func(N_Vector u, N_Vector f, void *f_data)
301  {
302    realtype *udata, *fdata;
303    realtype x1, l1, L1, x2, l2, L2;
304    realtype *lb, *ub;
305    UserData data;
306
307    data = (UserData)f_data;
308    lb = data->lb;
309    ub = data->ub;
310
311    udata = NV_DATA_S(u);
312    fdata = NV_DATA_S(f);
313
314    x1 = udata[0];
315    x2 = udata[1];
316    l1 = udata[2];
317    L1 = udata[3];
318    l2 = udata[4];
319    L2 = udata[5];
320
321    fdata[0] = PT5 * sin(x1*x2) - PT25 * x2 / PI - PT5 * x1;
322    fdata[1] = (ONE - PT25/PI)*(EXP(TWO*x1)-E) + E*x2/PI - TWO*E*x1;
323    fdata[2] = l1 - x1 + lb[0];
324    fdata[3] = L1 - x1 + ub[0];
325    fdata[4] = l2 - x2 + lb[1];
326    fdata[5] = L2 - x2 + ub[1];
327
328    return(0);
329  }
330
331  /*
332   *--------------------------------------------------------------------
333   * PRIVATE FUNCTIONS
334   *--------------------------------------------------------------------
335   */
336
337  /*
338   * Initial guesses
339   */
340
341  static void SetInitialGuess1(N_Vector u, UserData data)
342  {
343    realtype x1, x2;
344    realtype *udata;
345    realtype *lb, *ub;
346
347    udata = NV_DATA_S(u);
348
349    lb = data->lb;
350    ub = data->ub;
351
352    /* There are two known solutions for this problem */
```

```
353
354   /* this init. guess should take us to (0.29945; 2.83693) */
355   x1 = lb[0];
356   x2 = lb[1];
357
358   udata[0] = x1;
359   udata[1] = x2;
360   udata[2] = x1 - lb[0];
361   udata[3] = x1 - ub[0];
362   udata[4] = x2 - lb[1];
363   udata[5] = x2 - ub[1];
364 }
365
366 static void SetInitialGuess2(N_Vector u, UserData data)
367 {
368   realtype x1, x2;
369   realtype *udata;
370   realtype *lb, *ub;
371
372   udata = NV_DATA_S(u);
373
374   lb = data->lb;
375   ub = data->ub;
376
377   /* There are two known solutions for this problem */
378
379   /* this init. guess should take us to (0.5; 3.1415926) */
380   x1 = PT5 * (lb[0] + ub[0]);
381   x2 = PT5 * (lb[1] + ub[1]);
382
383   udata[0] = x1;
384   udata[1] = x2;
385   udata[2] = x1 - lb[0];
386   udata[3] = x1 - ub[0];
387   udata[4] = x2 - lb[1];
388   udata[5] = x2 - ub[1];
389 }
390
391 /*
392  * Print first lines of output (problem description)
393  */
394
395 static void PrintHeader(int globalstrategy, realtype fnormtol, realtype scsteptol)
396 {
397   printf("\nFerraris␣and␣Tronconi␣test␣problem\n");
398   printf("Tolerance␣parameters:\n");
399 #if defined(SUNDIALS_EXTENDED_PRECISION)
400   printf("␣␣fnormtol␣␣=␣%10.6Lg\n␣␣scsteptol␣=␣%10.6Lg\n",
401          fnormtol, scsteptol);
402 #elif defined(SUNDIALS_DOUBLE_PRECISION)
403   printf("␣␣fnormtol␣␣=␣%10.6lg\n␣␣scsteptol␣=␣%10.6lg\n",
404          fnormtol, scsteptol);
405 #else
406   printf("␣␣fnormtol␣␣=␣%10.6g\n␣␣scsteptol␣=␣%10.6g\n",
407          fnormtol, scsteptol);
408 #endif
409 }
410
411 /*
```

```
412   * Print solution
413   */
414
415  static void PrintOutput(N_Vector u)
416  {
417  #if defined(SUNDIALS_EXTENDED_PRECISION)
418      printf("␣%8.6Lg␣␣%8.6Lg\n", Ith(u,1), Ith(u,2));
419  #elif defined(SUNDIALS_DOUBLE_PRECISION)
420      printf("␣%8.6lg␣␣%8.6lg\n", Ith(u,1), Ith(u,2));
421  #else
422      printf("␣%8.6g␣␣%8.6g\n", Ith(u,1), Ith(u,2));
423  #endif
424  }
425
426  /*
427   * Print final statistics contained in iopt
428   */
429
430  static void PrintFinalStats(void *kmem)
431  {
432    long int nni, nfe, nje, nfeD;
433    int flag;
434
435    flag = KINGetNumNonlinSolvIters(kmem, &nni);
436    check_flag(&flag, "KINGetNumNonlinSolvIters", 1);
437    flag = KINGetNumFuncEvals(kmem, &nfe);
438    check_flag(&flag, "KINGetNumFuncEvals", 1);
439
440    flag = KINDenseGetNumJacEvals(kmem, &nje);
441    check_flag(&flag, "KINDenseGetNumJacEvals", 1);
442    flag = KINDenseGetNumFuncEvals(kmem, &nfeD);
443    check_flag(&flag, "KINDenseGetNumFuncEvals", 1);
444
445    printf("Final␣Statistics:\n");
446    printf("␣␣nni␣=␣%5ld␣␣␣␣nfe␣␣=␣%5ld␣\n", nni, nfe);
447    printf("␣␣nje␣=␣%5ld␣␣␣␣nfeD␣=␣%5ld␣\n", nje, nfeD);
448  }
449
450  /*
451   * Check function return value...
452   *    opt == 0 means SUNDIALS function allocates memory so check if
453   *             returned NULL pointer
454   *    opt == 1 means SUNDIALS function returns a flag so check if
455   *             flag >= 0
456   *    opt == 2 means function allocates memory so check if returned
457   *             NULL pointer
458   */
459
460  static int check_flag(void *flagvalue, char *funcname, int opt)
461  {
462    int *errflag;
463
464    /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
465    if (opt == 0 && flagvalue == NULL) {
466      fprintf(stderr,
467              "\nSUNDIALS_ERROR:␣%s()␣failed␣-␣returned␣NULL␣pointer\n\n",
468              funcname);
469      return(1);
470    }
```

```
471
472     /* Check if flag < 0 */
473     else if (opt == 1) {
474       errflag = (int *) flagvalue;
475       if (*errflag < 0) {
476         fprintf(stderr,
477                 "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
478                 funcname, *errflag);
479         return(1);
480       }
481     }
482
483     /* Check if function returned NULL pointer - no memory allocated */
484     else if (opt == 2 && flagvalue == NULL) {
485       fprintf(stderr,
486               "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
487               funcname);
488       return(1);
489     }
490
491     return(0);
492   }
```

B Listing of kinkryx.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.2 $
 * $Date: 2006/10/11 16:34:08 $
 * -----------------------------------------------------------------
 * Programmer(s): Allan Taylor, Alan Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example (serial):
 *
 * This example solves a nonlinear system that arises from a system
 * of partial differential equations. The PDE system is a food web
 * population model, with predator-prey interaction and diffusion
 * on the unit square in two dimensions. The dependent variable
 * vector is the following:
 *
 *       1   2         ns
 * c = (c , c ,  ..., c  )     (denoted by the variable cc)
 *
 * and the PDE's are as follows:
 *
 *                     i       i
 *         0 = d(i)*(c     + c    )  +  f  (x,y,c)   (i=1,...,ns)
 *                     xx      yy         i
 *
 *   where
 *
 *                   i             ns         j
 *   f  (x,y,c)  =  c  * (b(i)  + sum a(i,j)*c )
 *    i                            j=1
 *
 * The number of species is ns = 2 * np, with the first np being
 * prey and the last np being predators. The number np is both the
 * number of prey and predator species. The coefficients a(i,j),
 * b(i), d(i) are:
 *
 *   a(i,i) = -AA   (all i)
 *   a(i,j) = -GG   (i <= np , j >  np)
 *   a(i,j) =  EE   (i >  np,  j <= np)
 *   b(i) = BB * (1 + alpha * x * y)   (i <= np)
 *   b(i) =-BB * (1 + alpha * x * y)   (i >  np)
 *   d(i) = DPREY   (i <= np)
 *   d(i) = DPRED   ( i > np)
 *
 * The various scalar parameters are set using define's or in
 * routine InitUserData.
 *
 * The boundary conditions are: normal derivative = 0, and the
 * initial guess is constant in x and y, but the final solution
 * is not.
 *
 * The PDEs are discretized by central differencing on an MX by
 * MY mesh.
 *
 * The nonlinear system is solved by KINSOL using the method
 * specified in local variable globalstrat.
 *
```

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

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <kinsol/kinsol.h>
#include <kinsol/kinsol_spgmr.h>
#include <nvector/nvector_serial.h>
#include <sundials/sundials_smalldense.h>
#include <sundials/sundials_types.h>
#include <sundials/sundials_math.h>

/* 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 int func(N_Vector cc, N_Vector fval, void *f_data);

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

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

/* Private Helper Functions */

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

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

```
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_NONE;
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
220    /* Vector cc passed as template vector. */
221    flag = KINMalloc(kmem, func, cc);
222    if (check_flag(&flag, "KINMalloc", 1)) return(1);
223
224    flag = KINSetFdata(kmem, data);
225    if (check_flag(&flag, "KINSetFdata", 1)) return(1);
226    flag = KINSetConstraints(kmem, constraints);
227    if (check_flag(&flag, "KINSetConstraints", 1)) return(1);
228    flag = KINSetFuncNormTol(kmem, fnormtol);
229    if (check_flag(&flag, "KINSetFuncNormTol", 1)) return(1);
230    flag = KINSetScaledStepTol(kmem, scsteptol);
231    if (check_flag(&flag, "KINSetScaledStepTol", 1)) return(1);
232
233    /* We no longer need the constraints vector since KINSetConstraints
234       creates a private copy for KINSOL to use. */
```

```
235    N_VDestroy_Serial(constraints);
236
237    /* Call KINSpgmr to specify the linear solver KINSPGMR with preconditioner
238       routines PrecSetupBD and PrecSolveBD, and the pointer to the user block data. */
239    maxl = 15;
240    maxlrst = 2;
241    flag = KINSpgmr(kmem, maxl);
242    if (check_flag(&flag, "KINSpgmr", 1)) return(1);
243
244    flag = KINSpilsSetMaxRestarts(kmem, maxlrst);
245    if (check_flag(&flag, "KINSpilsSetMaxRestarts", 1)) return(1);
246    flag = KINSpilsSetPreconditioner(kmem,
247                                     PrecSetupBD,
248                                     PrecSolveBD,
249                                     data);
250    if (check_flag(&flag, "KINSpilsSetPreconditioner", 1)) return(1);
251
252    /* Print out the problem size, solution parameters, initial guess. */
253    PrintHeader(globalstrategy, maxl, maxlrst, fnormtol, scsteptol);
254
255    /* Call KINSol and print output concentration profile */
256    flag = KINSol(kmem,           /* KINSol memory block */
257                  cc,             /* initial guess on input; solution vector */
258                  globalstrategy, /* global stragegy choice */
259                  sc,             /* scaling vector, for the variable cc */
260                  sc);            /* scaling vector for function values fval */
261    if (check_flag(&flag, "KINSol", 1)) return(1);
262
263    printf("\n\nComputed␣equilibrium␣species␣concentrations:\n");
264    PrintOutput(cc);
265
266    /* Print final statistics and free memory */
267    PrintFinalStats(kmem);
268
269    N_VDestroy_Serial(cc);
270    N_VDestroy_Serial(sc);
271    KINFree(&kmem);
272    FreeUserData(data);
273
274    return(0);
275  }
276
277  /* Readability definitions used in other routines below */
278
279  #define acoef  (data->acoef)
280  #define bcoef  (data->bcoef)
281  #define cox    (data->cox)
282  #define coy    (data->coy)
283
284  /*
285   *--------------------------------------------------------------------
286   * FUNCTIONS CALLED BY KINSOL
287   *--------------------------------------------------------------------
288   */
289
290  /*
291   * System function for predator-prey system
292   */
293
```

```
294 static int func(N_Vector cc, N_Vector fval, void *f_data)
295 {
296   realtype xx, yy, delx, dely, *cxy, *rxy, *fxy, dcyli, dcyui, dcxli, dcxri;
297   long int jx, jy, is, idyu, idyl, idxr, idxl;
298   UserData data;
299
300   data = (UserData)f_data;
301   delx = data->dx;
302   dely = data->dy;
303
304   /* Loop over all mesh points, evaluating rate array at each point*/
305   for (jy = 0; jy < MY; jy++) {
306
307     yy = dely*jy;
308
309     /* Set lower/upper index shifts, special at boundaries. */
310     idyl = (jy != 0   ) ? NSMX : -NSMX;
311     idyu = (jy != MY-1) ? NSMX : -NSMX;
312
313     for (jx = 0; jx < MX; jx++) {
314
315       xx = delx*jx;
316
317       /* Set left/right index shifts, special at boundaries. */
318       idxl = (jx !=  0  ) ?  NUM_SPECIES : -NUM_SPECIES;
319       idxr = (jx != MX-1) ?  NUM_SPECIES : -NUM_SPECIES;
320
321       cxy = IJ_Vptr(cc,jx,jy);
322       rxy = IJ_Vptr(data->rates,jx,jy);
323       fxy = IJ_Vptr(fval,jx,jy);
324
325       /* Get species interaction rate array at (xx,yy) */
326       WebRate(xx, yy, cxy, rxy, f_data);
327
328       for(is = 0; is < NUM_SPECIES; is++) {
329
330         /* Differencing in x direction */
331         dcyli = *(cxy+is) - *(cxy - idyl + is) ;
332         dcyui = *(cxy + idyu + is) - *(cxy+is);
333
334         /* Differencing in y direction */
335         dcxli = *(cxy+is) - *(cxy - idxl + is);
336         dcxri = *(cxy + idxr +is) - *(cxy+is);
337
338         /* Compute the total rate value at (xx,yy) */
339         fxy[is] = (coy)[is] * (dcyui - dcyli) +
340           (cox)[is] * (dcxri - dcxli) + rxy[is];
341
342       } /* end of is loop */
343
344     } /* end of jx loop */
345
346   } /* end of jy loop */
347
348   return(0);
349 }
350
351 /*
352  * Preconditioner setup routine. Generate and preprocess P.
```

```
353   */
354
355  static int PrecSetupBD(N_Vector cc, N_Vector cscale,
356                         N_Vector fval, N_Vector fscale,
357                         void *P_data,
358                         N_Vector vtemp1, N_Vector vtemp2)
359  {
360    realtype r, r0, uround, sqruround, xx, yy, delx, dely, csave, fac;
361    realtype *cxy, *scxy, **Pxy, *ratesxy, *Pxycol, perturb_rates[NUM_SPECIES];
362    long int i, j, jx, jy, ret;
363    UserData data;
364
365    data = (UserData) P_data;
366    delx = data->dx;
367    dely = data->dy;
368
369    uround = data->uround;
370    sqruround = data->sqruround;
371    fac = N_VWL2Norm(fval, fscale);
372    r0 = THOUSAND * uround * fac * NEQ;
373    if(r0 == ZERO) r0 = ONE;
374
375    /* Loop over spatial points; get size NUM_SPECIES Jacobian block at each */
376    for (jy = 0; jy < MY; jy++) {
377      yy = jy*dely;
378
379      for (jx = 0; jx < MX; jx++) {
380        xx = jx*delx;
381        Pxy = (data->P)[jx][jy];
382        cxy = IJ_Vptr(cc,jx,jy);
383        scxy= IJ_Vptr(cscale,jx,jy);
384        ratesxy = IJ_Vptr((data->rates),jx,jy);
385
386        /* Compute difference quotients of interaction rate fn. */
387        for (j = 0; j < NUM_SPECIES; j++) {
388
389          csave = cxy[j];  /* Save the j,jx,jy element of cc */
390          r = MAX(sqruround*ABS(csave), r0/scxy[j]);
391          cxy[j] += r; /* Perturb the j,jx,jy element of cc */
392          fac = ONE/r;
393
394          WebRate(xx, yy, cxy, perturb_rates, data);
395
396          /* Restore j,jx,jy element of cc */
397          cxy[j] = csave;
398
399          /* Load the j-th column of difference quotients */
400          Pxycol = Pxy[j];
401          for (i = 0; i < NUM_SPECIES; i++)
402            Pxycol[i] = (perturb_rates[i] - ratesxy[i]) * fac;
403
404
405        } /* end of j loop */
406
407        /* Do LU decomposition of size NUM_SPECIES preconditioner block */
408        ret = denGETRF(Pxy, NUM_SPECIES, NUM_SPECIES, (data->pivot)[jx][jy]);
409        if (ret != 0) return(1);
410
411      } /* end of jx loop */
```

```
412
413     } /* end of jy loop */
414
415     return(0);
416   }
417
418   /*
419    * Preconditioner solve routine
420    */
421
422   static int PrecSolveBD(N_Vector cc, N_Vector cscale,
423                          N_Vector fval, N_Vector fscale,
424                          N_Vector vv, void *P_data,
425                          N_Vector ftem)
426   {
427     realtype **Pxy, *vxy;
428     long int *piv, jx, jy;
429     UserData data;
430
431     data = (UserData)P_data;
432
433     for (jx=0; jx<MX; jx++) {
434
435       for (jy=0; jy<MY; jy++) {
436
437         /* For each (jx,jy), solve a linear system of size NUM_SPECIES.
438            vxy is the address of the corresponding portion of the vector vv;
439            Pxy is the address of the corresponding block of the matrix P;
440            piv is the address of the corresponding block of the array pivot. */
441         vxy = IJ_Vptr(vv,jx,jy);
442         Pxy = (data->P)[jx][jy];
443         piv = (data->pivot)[jx][jy];
444         denGETRS(Pxy, NUM_SPECIES, piv, vxy);
445
446       } /* end of jy loop */
447
448     } /* end of jx loop */
449
450     return(0);
451   }
452
453   /*
454    * Interaction rate function routine
455    */
456
457   static void WebRate(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
458                       void *f_data)
459   {
460     long int i;
461     realtype fac;
462     UserData data;
463
464     data = (UserData)f_data;
465
466     for (i = 0; i<NUM_SPECIES; i++)
467       ratesxy[i] = DotProd(NUM_SPECIES, cxy, acoef[i]);
468
469     fac = ONE + ALPHA * xx * yy;
470
```

```
471     for (i = 0; i < NUM_SPECIES; i++)
472       ratesxy[i] = cxy[i] * ( bcoef[i] * fac + ratesxy[i] );
473   }
474
475   /*
476    * Dot product routine for realtype arrays
477    */
478
479   static realtype DotProd(long int size, realtype *x1, realtype *x2)
480   {
481     long int i;
482     realtype *xx1, *xx2, temp = ZERO;
483
484     xx1 = x1; xx2 = x2;
485     for (i = 0; i < size; i++) temp += (*xx1++) * (*xx2++);
486
487     return(temp);
488   }
489
490   /*
491    *--------------------------------------------------------------------
492    * PRIVATE FUNCTIONS
493    *--------------------------------------------------------------------
494    */
495
496   /*
497    * Allocate memory for data structure of type UserData
498    */
499
500   static UserData AllocUserData(void)
501   {
502     int jx, jy;
503     UserData data;
504
505     data = (UserData) malloc(sizeof *data);
506
507     for (jx=0; jx < MX; jx++) {
508       for (jy=0; jy < MY; jy++) {
509         (data->P)[jx][jy] = denalloc(NUM_SPECIES, NUM_SPECIES);
510         (data->pivot)[jx][jy] = denallocpiv(NUM_SPECIES);
511       }
512     }
513     acoef = denalloc(NUM_SPECIES, NUM_SPECIES);
514     bcoef = (realtype *)malloc(NUM_SPECIES * sizeof(realtype));
515     cox   = (realtype *)malloc(NUM_SPECIES * sizeof(realtype));
516     coy   = (realtype *)malloc(NUM_SPECIES * sizeof(realtype));
517
518     return(data);
519   }
520
521   /*
522    * Load problem constants in data
523    */
524
525   static void InitUserData(UserData data)
526   {
527     long int i, j, np;
528     realtype *a1,*a2, *a3, *a4, dx2, dy2;
529
```

```
530     data->mx = MX;
531     data->my = MY;
532     data->ns = NUM_SPECIES;
533     data->np = NUM_SPECIES/2;
534     data->ax = AX;
535     data->ay = AY;
536     data->dx = (data->ax)/(MX-1);
537     data->dy = (data->ay)/(MY-1);
538     data->uround = UNIT_ROUNDOFF;
539     data->sqruround = SQRT(data->uround);
540
541     /* Set up the coefficients a and b plus others found in the equations */
542     np = data->np;
543
544     dx2=(data->dx)*(data->dx); dy2=(data->dy)*(data->dy);
545
546     for (i = 0; i < np; i++) {
547       a1= &(acoef[i][np]);
548       a2= &(acoef[i+np][0]);
549       a3= &(acoef[i][0]);
550       a4= &(acoef[i+np][np]);
551
552       /*  Fill in the portion of acoef in the four quadrants, row by row */
553       for (j = 0; j < np; j++) {
554         *a1++ =  -GG;
555         *a2++ =   EE;
556         *a3++ = ZERO;
557         *a4++ = ZERO;
558       }
559
560       /* and then change the diagonal elements of acoef to -AA */
561       acoef[i][i]=-AA;
562       acoef[i+np][i+np] = -AA;
563
564       bcoef[i] = BB;
565       bcoef[i+np] = -BB;
566
567       cox[i]=DPREY/dx2;
568       cox[i+np]=DPRED/dx2;
569
570       coy[i]=DPREY/dy2;
571       coy[i+np]=DPRED/dy2;
572     }
573   }
574
575   /*
576    * Free data memory
577    */
578
579   static void FreeUserData(UserData data)
580   {
581     int jx, jy;
582
583     for (jx=0; jx < MX; jx++) {
584       for (jy=0; jy < MY; jy++) {
585         denfree((data->P)[jx][jy]);
586         denfreepiv((data->pivot)[jx][jy]);
587       }
588     }
```

```
589
590   denfree(acoef);
591   free(bcoef);
592   free(cox);
593   free(coy);
594   N_VDestroy_Serial(data->rates);
595   free(data);
596 }
597
598 /*
599  * Set initial conditions in cc
600  */
601
602 static void SetInitialProfiles(N_Vector cc, N_Vector sc)
603 {
604   int i, jx, jy;
605   realtype *cloc, *sloc;
606   realtype  ctemp[NUM_SPECIES], stemp[NUM_SPECIES];
607
608   /* Initialize arrays ctemp and stemp used in the loading process */
609   for (i = 0; i < NUM_SPECIES/2; i++) {
610     ctemp[i] = PREYIN;
611     stemp[i] = ONE;
612   }
613   for (i = NUM_SPECIES/2; i < NUM_SPECIES; i++) {
614     ctemp[i] = PREDIN;
615     stemp[i] = RCONST(0.00001);
616   }
617
618   /* Load initial profiles into cc and sc vector from ctemp and stemp. */
619   for (jy = 0; jy < MY; jy++) {
620     for (jx = 0; jx < MX; jx++) {
621       cloc = IJ_Vptr(cc,jx,jy);
622       sloc = IJ_Vptr(sc,jx,jy);
623       for (i = 0; i < NUM_SPECIES; i++) {
624         cloc[i] = ctemp[i];
625         sloc[i] = stemp[i];
626       }
627     }
628   }
629 }
630
631 /*
632  * Print first lines of output (problem description)
633  */
634
635 static void PrintHeader(int globalstrategy, int maxl, int maxlrst,
636                         realtype fnormtol, realtype scsteptol)
637 {
638   printf("\nPredator-prey test problem -- KINSol (serial version)\n\n");
639   printf("Mesh dimensions = %d X %d\n", MX, MY);
640   printf("Number of species = %d\n", NUM_SPECIES);
641   printf("Total system size = %d\n\n", NEQ);
642   printf("Flag globalstrategy = %d (0 = None, 1 = Linesearch)\n",
643          globalstrategy);
644   printf("Linear solver is SPGMR with maxl = %d, maxlrst = %d\n",
645          maxl, maxlrst);
646   printf("Preconditioning uses interaction-only block-diagonal matrix\n");
647   printf("Positivity constraints imposed on all components \n");
```

```
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("Tolerance␣parameters:␣␣fnormtol␣=␣%Lg␣␣␣scsteptol␣=␣%Lg\n",
         fnormtol, scsteptol);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("Tolerance␣parameters:␣␣fnormtol␣=␣%lg␣␣␣scsteptol␣=␣%lg\n",
         fnormtol, scsteptol);
#else
  printf("Tolerance␣parameters:␣␣fnormtol␣=␣%g␣␣␣scsteptol␣=␣%g\n",
         fnormtol, scsteptol);
#endif

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

/*
 * Print sampled values of current cc
 */

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

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

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

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

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

```
#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf("␣%Lg",ct[is]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
    printf("␣%lg",ct[is]);
#else
    printf("␣%g",ct[is]);
#endif
  }
  printf("\n\n");
}

/*
 * Print final statistics contained in iopt
 */

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

  flag = KINGetNumNonlinSolvIters(kmem, &nni);
  check_flag(&flag, "KINGetNumNonlinSolvIters", 1);
  flag = KINGetNumFuncEvals(kmem, &nfe);
  check_flag(&flag, "KINGetNumFuncEvals", 1);
  flag = KINSpilsGetNumLinIters(kmem, &nli);
  check_flag(&flag, "KINSpilsGetNumLinIters", 1);
  flag = KINSpilsGetNumPrecEvals(kmem, &npe);
  check_flag(&flag, "KINSpilsGetNumPrecEvals", 1);
  flag = KINSpilsGetNumPrecSolves(kmem, &nps);
  check_flag(&flag, "KINSpilsGetNumPrecSolves", 1);
  flag = KINSpilsGetNumConvFails(kmem, &ncfl);
  check_flag(&flag, "KINSpilsGetNumConvFails", 1);
  flag = KINSpilsGetNumFuncEvals(kmem, &nfeSG);
  check_flag(&flag, "KINSpilsGetNumFuncEvals", 1);

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

}

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

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

  /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
  if (opt == 0 && flagvalue == NULL) {
    fprintf(stderr,
```

```
766               "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
767               funcname);
768       return(1);
769     }
770
771     /* Check if flag < 0 */
772     else if (opt == 1) {
773       errflag = (int *) flagvalue;
774       if (*errflag < 0) {
775         fprintf(stderr,
776                 "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
777                 funcname, *errflag);
778         return(1);
779       }
780     }
781
782     /* Check if function returned NULL pointer - no memory allocated */
783     else if (opt == 2 && flagvalue == NULL) {
784       fprintf(stderr,
785               "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
786               funcname);
787       return(1);
788     }
789
790     return(0);
791   }
```

C Listing of kinkryx_bbd_p.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.2 $
 * $Date: 2006/10/11 16:34:06 $
 * -----------------------------------------------------------------
 * 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 as follows:
 *
 *   Difference quotient half-bandwidths mldq = mudq = 2*ns - 1
 *   Retained banded blocks have half-bandwidths mlkeep = mukeep = ns.
 *
 * -----------------------------------------------------------------
 * 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 kinkryx_bbd_p
 *  where N = NPEX * NPEY is the number of processors.
 * -----------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <kinsol/kinsol.h>
#include <kinsol/kinsol_spgmr.h>
#include <kinsol/kinsol_bbdpre.h>
#include <nvector/nvector_parallel.h>
#include <sundials/sundials_smalldense.h>
#include <sundials/sundials_math.h>
#include <sundials/sundials_types.h>

#include <mpi.h>


/* 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 int func(N_Vector cc, N_Vector fval, void *f_data);

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

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

/* Private Helper Functions */

static UserData AllocUserData(void);
static void InitUserData(long int my_pe, long int Nlocal, MPI_Comm comm, UserData data);
static void FreeUserData(UserData data);
static void SetInitialProfiles(N_Vector cc, N_Vector sc);
static void PrintHeader(int globalstrategy, int maxl, int maxlrst,
                        long int mudq, long int mldq,
                        long int mukeep, long int mlkeep,
                        realtype fnormtol, realtype scsteptol);
static void PrintOutput(long int my_pe, MPI_Comm comm, N_Vector cc);
static void PrintFinalStats(void *kmem);
```

```
176 static void WebRate(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
177                      void *f_data);
178 static realtype DotProd(long int size, realtype *x1, realtype *x2);
179 static void BSend(MPI_Comm comm, long int my_pe, long int isubx,
180                   long int isuby, long int dsizex, long int dsizey,
181                   realtype *cdata);
182 static void BRecvPost(MPI_Comm comm, MPI_Request request[], long int my_pe,
183                       long int isubx, long int isuby,
184                       long int dsizex, long int dsizey,
185                       realtype *cext, realtype *buffer);
186 static void BRecvWait(MPI_Request request[], long int isubx,
187                       long int isuby, long int dsizex, realtype *cext,
188                       realtype *buffer);
189 static int check_flag(void *flagvalue, char *funcname, int opt, int id);
190
191 /*
192  *--------------------------------------------------------------------
193  * MAIN PROGRAM
194  *--------------------------------------------------------------------
195  */
196
197 int main(int argc, char *argv[])
198 {
199   MPI_Comm comm;
200   void *kmem, *pdata;
201   UserData data;
202   N_Vector cc, sc, constraints;
203   int globalstrategy;
204   long int Nlocal;
205   realtype fnormtol, scsteptol, dq_rel_uu;
206   int flag, maxl, maxlrst;
207   long int mudq, mldq, mukeep, mlkeep;
208   int my_pe, npes, npelast = NPEX*NPEY-1;
209
210   data = NULL;
211   kmem = pdata = NULL;
212   cc = sc = constraints = NULL;
213
214   /* Get processor number and total number of pe's */
215   MPI_Init(&argc, &argv);
216   comm = MPI_COMM_WORLD;
217   MPI_Comm_size(comm, &npes);
218   MPI_Comm_rank(comm, &my_pe);
219
220   if (npes != NPEX*NPEY) {
221     if (my_pe == 0)
222       printf("\nMPI_ERROR(0): npes=%d is not equal to NPEX*NPEY=%d\n", npes, NPEX*NPEY);
223     return(1);
224   }
225
226   /* Allocate memory, and set problem data, initial values, tolerances */
227
228   /* Set local length */
229   Nlocal = NUM_SPECIES*MXSUB*MYSUB;
230
231   /* Allocate and initialize user data block */
232   data = AllocUserData();
233   if (check_flag((void *)data, "AllocUserData", 2, my_pe)) MPI_Abort(comm, 1);
234   InitUserData(my_pe, Nlocal, comm, data);
```

```
235
236     /* Choose global strategy */
237     globalstrategy = KIN_NONE;
238
239     /* Allocate and initialize vectors */
240     cc = N_VNew_Parallel(comm, Nlocal, NEQ);
241     if (check_flag((void *)cc, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
242     sc = N_VNew_Parallel(comm, Nlocal, NEQ);
243     if (check_flag((void *)sc, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
244     data->rates = N_VNew_Parallel(comm, Nlocal, NEQ);
245     if (check_flag((void *)data->rates, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
246     constraints = N_VNew_Parallel(comm, Nlocal, NEQ);
247     if (check_flag((void *)constraints, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
248     N_VConst(ZERO, constraints);
249
250     SetInitialProfiles(cc, sc);
251
252     fnormtol = FTOL; scsteptol = STOL;
253
254     /* Call KINCreate/KINMalloc to initialize KINSOL:
255        nvSpec  points to machine environment data
256        A pointer to KINSOL problem memory is returned and stored in kmem. */
257     kmem = KINCreate();
258     if (check_flag((void *)kmem, "KINCreate", 0, my_pe)) MPI_Abort(comm, 1);
259
260     /* Vector cc passed as template vector. */
261     flag = KINMalloc(kmem, func, cc);
262     if (check_flag(&flag, "KINMalloc", 1, my_pe)) MPI_Abort(comm, 1);
263
264     flag = KINSetFdata(kmem, data);
265     if (check_flag(&flag, "KINSetFdata", 1, my_pe)) MPI_Abort(comm, 1);
266
267     flag = KINSetConstraints(kmem, constraints);
268     if (check_flag(&flag, "KINSetConstraints", 1, my_pe)) MPI_Abort(comm, 1);
269
270     /* We no longer need the constraints vector since KINSetConstraints
271        creates a private copy for KINSOL to use. */
272     N_VDestroy_Parallel(constraints);
273
274     flag = KINSetFuncNormTol(kmem, fnormtol);
275     if (check_flag(&flag, "KINSetFuncNormTol", 1, my_pe)) MPI_Abort(comm, 1);
276
277     flag = KINSetScaledStepTol(kmem, scsteptol);
278     if (check_flag(&flag, "KINSetScaledStepTol", 1, my_pe)) MPI_Abort(comm, 1);
279
280     /* Call KINBBDPrecAlloc to initialize and allocate memory for the
281        band-block-diagonal preconditioner, and specify the local and
282        communication functions func_local and gcomm=NULL (all communication
283        needed for the func_local is already done in func). */
284     dq_rel_uu = ZERO;
285     mudq = mldq = 2*NUM_SPECIES - 1;
286     mukeep = mlkeep = NUM_SPECIES;
287
288     pdata = KINBBDPrecAlloc(kmem, Nlocal, mudq, mldq, mukeep, mlkeep,
289                             dq_rel_uu, func_local, NULL);
290     if (check_flag((void *)pdata, "KINBBDPrecAlloc", 0, my_pe))
291       MPI_Abort(comm, 1);
292
293     /* Call KINBBDSpgmr to specify the linear solver KINSPGMR
```

```
294       with preconditioner KINBBDPRE */
295     maxl = 20; maxlrst = 2;
296     flag = KINBBDSpgmr(kmem, maxl, pdata);
297     if (check_flag(&flag, "KINBBDSpgmr", 1, my_pe))
298       MPI_Abort(comm, 1);
299
300     flag = KINSpilsSetMaxRestarts(kmem, maxlrst);
301     if (check_flag(&flag, "KINSpilsSetMaxRestarts", 1, my_pe))
302       MPI_Abort(comm, 1);
303
304     /* Print out the problem size, solution parameters, initial guess. */
305     if (my_pe == 0)
306       PrintHeader(globalstrategy, maxl, maxlrst, mudq, mldq, mukeep,
307                   mlkeep, fnormtol, scsteptol);
308
309     /* call KINSol and print output concentration profile */
310     flag = KINSol(kmem,           /* KINSol memory block */
311                   cc,             /* initial guesss on input; solution vector */
312                   globalstrategy, /* global stragegy choice */
313                   sc,             /* scaling vector, for the variable cc */
314                   sc);            /* scaling vector for function values fval */
315     if (check_flag(&flag, "KINSol", 1, my_pe)) MPI_Abort(comm, 1);
316
317     if (my_pe == 0) printf("\n\nComputed␣equilibrium␣species␣concentrations:\n");
318     if (my_pe == 0 || my_pe==npelast) PrintOutput(my_pe, comm, cc);
319
320     /* Print final statistics and free memory */
321     if (my_pe == 0)
322       PrintFinalStats(kmem);
323
324     N_VDestroy_Parallel(cc);
325     N_VDestroy_Parallel(sc);
326     KINBBDPrecFree(&pdata);
327     KINFree(&kmem);
328     FreeUserData(data);
329
330     MPI_Finalize();
331
332     return(0);
333   }
334
335   /* Readability definitions used in other routines below */
336
337   #define acoef  (data->acoef)
338   #define bcoef  (data->bcoef)
339   #define cox    (data->cox)
340   #define coy    (data->coy)
341
342   /*
343    *--------------------------------------------------------------------
344    * FUNCTIONS CALLED BY KINSOL
345    *--------------------------------------------------------------------
346    */
347
348   /*
349    * ccomm routine.  This routine performs all communication
350    * between processors of data needed to calculate f.
351    */
352
```

```
353 static int ccomm(long int Nlocal, N_Vector cc, void *userdata)
354 {
355
356   realtype *cdata, *cext, buffer[2*NUM_SPECIES*MYSUB];
357   UserData data;
358   MPI_Comm comm;
359   long int my_pe, isubx, isuby, nsmxsub, nsmysub;
360   MPI_Request request[4];
361
362   /* Get comm, my_pe, subgrid indices, data sizes, extended array cext */
363   data = (UserData) userdata;
364   comm = data->comm;  my_pe = data->my_pe;
365   isubx = data->isubx;   isuby = data->isuby;
366   nsmxsub = data->nsmxsub;
367   nsmysub = NUM_SPECIES*MYSUB;
368   cext = data->cext;
369
370   cdata = NV_DATA_P(cc);
371
372   /* Start receiving boundary data from neighboring PEs */
373   BRecvPost(comm, request, my_pe, isubx, isuby, nsmxsub, nsmysub, cext, buffer);
374
375   /* Send data from boundary of local grid to neighboring PEs */
376   BSend(comm, my_pe, isubx, isuby, nsmxsub, nsmysub, cdata);
377
378   /* Finish receiving boundary data from neighboring PEs */
379   BRecvWait(request, isubx, isuby, nsmxsub, cext, buffer);
380
381   return(0);
382 }
383
384 /*
385  * System function for predator-prey system - calculation part
386  */
387
388 static int func_local(long int Nlocal, N_Vector cc, N_Vector fval, void *f_data)
389 {
390   realtype xx, yy, *cxy, *rxy, *fxy, dcydi, dcyui, dcxli, dcxri;
391   realtype *cext, dely, delx, *cdata;
392   long int i, jx, jy, is, ly;
393   long int isubx, isuby, nsmxsub, nsmxsub2;
394   long int shifty, offsetc, offsetce, offsetcl, offsetcr, offsetcd, offsetcu;
395   UserData data;
396
397   data = (UserData)f_data;
398   cdata = NV_DATA_P(cc);
399
400   /* Get subgrid indices, data sizes, extended work array cext */
401   isubx = data->isubx;   isuby = data->isuby;
402   nsmxsub = data->nsmxsub; nsmxsub2 = data->nsmxsub2;
403   cext = data->cext;
404
405   /* Copy local segment of cc vector into the working extended array cext */
406   offsetc = 0;
407   offsetce = nsmxsub2 + NUM_SPECIES;
408   for (ly = 0; ly < MYSUB; ly++) {
409     for (i = 0; i < nsmxsub; i++) cext[offsetce+i] = cdata[offsetc+i];
410     offsetc = offsetc + nsmxsub;
411     offsetce = offsetce + nsmxsub2;
```

```
412     }
413
414     /* To facilitate homogeneous Neumann boundary conditions, when this is a
415        boundary PE, copy data from the first interior mesh line of cc to cext */
416
417     /* If isuby = 0, copy x-line 2 of cc to cext */
418     if (isuby == 0) {
419       for (i = 0; i < nsmxsub; i++) cext[NUM_SPECIES+i] = cdata[nsmxsub+i];
420     }
421
422     /* If isuby = NPEY-1, copy x-line MYSUB-1 of cc to cext */
423     if (isuby == NPEY-1) {
424       offsetc = (MYSUB-2)*nsmxsub;
425       offsetce = (MYSUB+1)*nsmxsub2 + NUM_SPECIES;
426       for (i = 0; i < nsmxsub; i++) cext[offsetce+i] = cdata[offsetc+i];
427     }
428
429     /* If isubx = 0, copy y-line 2 of cc to cext */
430     if (isubx == 0) {
431       for (ly = 0; ly < MYSUB; ly++) {
432         offsetc = ly*nsmxsub + NUM_SPECIES;
433         offsetce = (ly+1)*nsmxsub2;
434         for (i = 0; i < NUM_SPECIES; i++) cext[offsetce+i] = cdata[offsetc+i];
435       }
436     }
437
438     /* If isubx = NPEX-1, copy y-line MXSUB-1 of cc to cext */
439     if (isubx == NPEX-1) {
440       for (ly = 0; ly < MYSUB; ly++) {
441         offsetc = (ly+1)*nsmxsub - 2*NUM_SPECIES;
442         offsetce = (ly+2)*nsmxsub2 - NUM_SPECIES;
443         for (i = 0; i < NUM_SPECIES; i++) cext[offsetce+i] = cdata[offsetc+i];
444       }
445     }
446
447     /* Loop over all mesh points, evaluating rate arra at each point */
448     delx = data->dx;
449     dely = data->dy;
450     shifty = (MXSUB+2)*NUM_SPECIES;
451
452     for (jy = 0; jy < MYSUB; jy++) {
453
454       yy = dely*(jy + isuby * MYSUB);
455
456       for (jx = 0; jx < MXSUB; jx++) {
457
458         xx = delx * (jx + isubx * MXSUB);
459         cxy = IJ_Vptr(cc,jx,jy);
460         rxy = IJ_Vptr(data->rates,jx,jy);
461         fxy = IJ_Vptr(fval,jx,jy);
462
463         WebRate(xx, yy, cxy, rxy, f_data);
464
465         offsetc = (jx+1)*NUM_SPECIES + (jy+1)*NSMXSUB2;
466         offsetcd = offsetc - shifty;
467         offsetcu = offsetc + shifty;
468         offsetcl = offsetc - NUM_SPECIES;
469         offsetcr = offsetc + NUM_SPECIES;
470
```

```
471         for (is = 0; is < NUM_SPECIES; is++) {
472
473           /* differencing in x */
474           dcydi = cext[offsetc+is]  - cext[offsetcd+is];
475           dcyui = cext[offsetcu+is] - cext[offsetc+is];
476
477           /* differencing in y */
478           dcxli = cext[offsetc+is]  - cext[offsetcl+is];
479           dcxri = cext[offsetcr+is] - cext[offsetc+is];
480
481           /* compute the value at xx , yy */
482           fxy[is] = (coy)[is] * (dcyui - dcydi) +
483             (cox)[is] * (dcxri - dcxli) + rxy[is];
484
485         } /* end of is loop */
486
487       } /* end of jx loop */
488
489     } /* end of jy loop */
490
491     return(0);
492   }
493
494   /*
495    * System function routine.  Evaluate f(cc).  First call ccomm to do
496    * communication of subgrid boundary data into cext.  Then calculate f
497    * by a call to func_local.
498    */
499
500   static int func(N_Vector cc, N_Vector fval, void *f_data)
501   {
502     UserData data;
503
504     data = (UserData) f_data;
505
506     /* Call ccomm to do inter-processor communicaiton */
507     ccomm(data->Nlocal, cc, data);
508
509     /* Call func_local to calculate all right-hand sides */
510     func_local(data->Nlocal, cc, fval, data);
511
512     return(0);
513   }
514
515   /*
516    * Interaction rate function routine
517    */
518
519   static void WebRate(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
520                       void *f_data)
521   {
522     long int i;
523     realtype fac;
524     UserData data;
525
526     data = (UserData)f_data;
527
528     for (i = 0; i<NUM_SPECIES; i++)
529       ratesxy[i] = DotProd(NUM_SPECIES, cxy, acoef[i]);
```

```
530
531     fac = ONE + ALPHA * xx * yy;
532
533     for (i = 0; i < NUM_SPECIES; i++)
534       ratesxy[i] = cxy[i] * ( bcoef[i] * fac + ratesxy[i] );
535   }
536
537   /*
538    * Dot product routine for realtype arrays
539    */
540
541   static realtype DotProd(long int size, realtype *x1, realtype *x2)
542   {
543     long int i;
544     realtype *xx1, *xx2, temp = ZERO;
545
546     xx1 = x1; xx2 = x2;
547     for (i = 0; i < size; i++) temp += (*xx1++) * (*xx2++);
548
549     return(temp);
550   }
551
552   /*
553    *--------------------------------------------------------------------
554    * PRIVATE FUNCTIONS
555    *--------------------------------------------------------------------
556    */
557
558   /*
559    * Allocate memory for data structure of type UserData
560    */
561
562   static UserData AllocUserData(void)
563   {
564     UserData data;
565
566     data = (UserData) malloc(sizeof *data);
567
568     acoef = denalloc(NUM_SPECIES, NUM_SPECIES);
569     bcoef = (realtype *)malloc(NUM_SPECIES * sizeof(realtype));
570     cox   = (realtype *)malloc(NUM_SPECIES * sizeof(realtype));
571     coy   = (realtype *)malloc(NUM_SPECIES * sizeof(realtype));
572
573     return(data);
574   }
575
576   /*
577    * Load problem constants in data
578    */
579
580   static void InitUserData(long int my_pe, long int Nlocal, MPI_Comm comm, UserData data)
581   {
582     long int i, j, np;
583     realtype *a1,*a2, *a3, *a4, dx2, dy2;
584
585     data->mx = MX;
586     data->my = MY;
587     data->ns = NUM_SPECIES;
588     data->np = NUM_SPECIES/2;
```

```
589     data->ax = AX;
590     data->ay = AY;
591     data->dx = (data->ax)/(MX-1);
592     data->dy = (data->ay)/(MY-1);
593     data->my_pe = my_pe;
594     data->Nlocal = Nlocal;
595     data->comm = comm;
596     data->isuby = my_pe/NPEX;
597     data->isubx = my_pe - data->isuby*NPEX;
598     data->nsmxsub = NUM_SPECIES * MXSUB;
599     data->nsmxsub2 = NUM_SPECIES * (MXSUB+2);
600
601     /* Set up the coefficients a and b plus others found in the equations */
602     np = data->np;
603
604     dx2=(data->dx)*(data->dx); dy2=(data->dy)*(data->dy);
605
606     for (i = 0; i < np; i++) {
607       a1= &(acoef[i][np]);
608       a2= &(acoef[i+np][0]);
609       a3= &(acoef[i][0]);
610       a4= &(acoef[i+np][np]);
611
612       /*  Fill in the portion of acoef in the four quadrants, row by row */
613       for (j = 0; j < np; j++) {
614         *a1++ =  -GG;
615         *a2++ =   EE;
616         *a3++ = ZERO;
617         *a4++ = ZERO;
618       }
619
620       /* and then change the diagonal elements of acoef to -AA */
621       acoef[i][i]=-AA;
622       acoef[i+np][i+np] = -AA;
623
624       bcoef[i] = BB;
625       bcoef[i+np] = -BB;
626
627       cox[i]=DPREY/dx2;
628       cox[i+np]=DPRED/dx2;
629
630       coy[i]=DPREY/dy2;
631       coy[i+np]=DPRED/dy2;
632     }
633   }
634
635   /*
636    * Free data memory
637    */
638
639   static void FreeUserData(UserData data)
640   {
641
642     denfree(acoef);
643     free(bcoef);
644     free(cox); free(coy);
645     N_VDestroy_Parallel(data->rates);
646
647     free(data);
```

```
648
649 }
650
651 /*
652  * Set initial conditions in cc
653  */
654
655 static void SetInitialProfiles(N_Vector cc, N_Vector sc)
656 {
657   int i, jx, jy;
658   realtype *cloc, *sloc;
659   realtype  ctemp[NUM_SPECIES], stemp[NUM_SPECIES];
660
661   /* Initialize arrays ctemp and stemp used in the loading process */
662   for (i = 0; i < NUM_SPECIES/2; i++) {
663     ctemp[i] = PREYIN;
664     stemp[i] = ONE;
665   }
666   for (i = NUM_SPECIES/2; i < NUM_SPECIES; i++) {
667     ctemp[i] = PREDIN;
668     stemp[i] = RCONST(0.00001);
669   }
670
671   /* Load initial profiles into cc and sc vector from ctemp and stemp. */
672   for (jy = 0; jy < MYSUB; jy++) {
673     for (jx=0; jx < MXSUB; jx++) {
674       cloc = IJ_Vptr(cc,jx,jy);
675       sloc = IJ_Vptr(sc,jx,jy);
676       for (i = 0; i < NUM_SPECIES; i++){
677         cloc[i] = ctemp[i];
678         sloc[i] = stemp[i];
679       }
680     }
681   }
682
683 }
684
685 /*
686  * Print first lines of output (problem description)
687  */
688
689 static void PrintHeader(int globalstrategy, int maxl, int maxlrst,
690                         long int mudq, long int mldq,
691                         long int mukeep, long int mlkeep,
692                         realtype fnormtol, realtype scsteptol)
693 {
694     printf("\nPredator-prey test problem --  KINSol (parallel-BBD version)\n\n");
695
696     printf("Mesh dimensions = %d X %d\n", MX, MY);
697     printf("Number of species = %d\n", NUM_SPECIES);
698     printf("Total system size = %d\n\n", NEQ);
699     printf("Subgrid dimensions = %d X %d\n", MXSUB, MYSUB);
700     printf("Processor array is %d X %d\n\n", NPEX, NPEY);
701     printf("Flag globalstrategy = %d (0 = None, 1 = Linesearch)\n",
702            globalstrategy);
703     printf("Linear solver is SPGMR with maxl = %d, maxlrst = %d\n",
704            maxl, maxlrst);
705     printf("Preconditioning uses band-block-diagonal matrix from KINBBDPRE\n");
706     printf("  Difference quotient half-bandwidths are mudq = %ld, mldq = %ld\n",
```

```
707             mudq, mldq);
708     printf("␣␣Retained␣band␣block␣half-bandwidths␣are␣mukeep␣=␣%ld,␣mlkeep␣=␣%ld\n",
709             mukeep, mlkeep);
710 #if defined(SUNDIALS_EXTENDED_PRECISION)
711     printf("Tolerance␣parameters:␣␣fnormtol␣=␣%Lg␣␣␣scsteptol␣=␣%Lg\n",
712             fnormtol, scsteptol);
713 #elif defined(SUNDIALS_DOUBLE_PRECISION)
714     printf("Tolerance␣parameters:␣␣fnormtol␣=␣%lg␣␣␣scsteptol␣=␣%lg\n",
715             fnormtol, scsteptol);
716 #else
717     printf("Tolerance␣parameters:␣␣fnormtol␣=␣%g␣␣␣scsteptol␣=␣%g\n",
718             fnormtol, scsteptol);
719 #endif
720
721     printf("\nInitial␣profile␣of␣concentration\n");
722 #if defined(SUNDIALS_EXTENDED_PRECISION)
723     printf("At␣all␣mesh␣points:␣␣%Lg␣%Lg␣%Lg␣␣␣%Lg␣%Lg␣%Lg\n", PREYIN,PREYIN,PREYIN,
724             PREDIN,PREDIN,PREDIN);
725 #elif defined(SUNDIALS_DOUBLE_PRECISION)
726     printf("At␣all␣mesh␣points:␣␣%lg␣%lg␣%lg␣␣␣%lg␣%lg␣%lg\n", PREYIN,PREYIN,PREYIN,
727             PREDIN,PREDIN,PREDIN);
728 #else
729     printf("At␣all␣mesh␣points:␣␣%g␣%g␣%g␣␣␣%g␣%g␣%g\n", PREYIN,PREYIN,PREYIN,
730             PREDIN,PREDIN,PREDIN);
731 #endif
732 }
733
734 /*
735  * Print sample of current cc values
736  */
737
738 static void PrintOutput(long int my_pe, MPI_Comm comm, N_Vector cc)
739 {
740   int is, i0, npelast;
741   realtype  *ct, tempc[NUM_SPECIES];
742   MPI_Status status;
743
744   npelast = NPEX*NPEY - 1;
745
746   ct = NV_DATA_P(cc);
747
748   /* Send the cc values (for all species) at the top right mesh point to PE 0 */
749   if (my_pe == npelast) {
750     i0 = NUM_SPECIES*(MXSUB*MYSUB-1);
751     if (npelast!=0)
752       MPI_Send(&ct[i0],NUM_SPECIES,PVEC_REAL_MPI_TYPE,0,0,comm);
753     else  /* single processor case */
754       for (is = 0; is < NUM_SPECIES; is++) tempc[is]=ct[i0+is];
755   }
756
757   /* On PE 0, receive the cc values at top right, then print performance data
758      and sampled solution values */
759   if (my_pe == 0) {
760
761     if (npelast != 0)
762       MPI_Recv(&tempc[0],NUM_SPECIES,PVEC_REAL_MPI_TYPE,npelast,0,comm,&status);
763
764     printf("\nAt␣bottom␣left:");
765     for (is = 0; is < NUM_SPECIES; is++){
```

```
766         if ((is%6)*6== is) printf("\n");
767   #if defined(SUNDIALS_EXTENDED_PRECISION)
768         printf("␣%Lg",ct[is]);
769   #elif defined(SUNDIALS_DOUBLE_PRECISION)
770         printf("␣%lg",ct[is]);
771   #else
772         printf("␣%g",ct[is]);
773   #endif
774       }
775
776       printf("\n\nAt␣top␣right:");
777       for (is = 0; is < NUM_SPECIES; is++) {
778         if ((is%6)*6 == is) printf("\n");
779   #if defined(SUNDIALS_EXTENDED_PRECISION)
780         printf("␣%Lg",tempc[is]);
781   #elif defined(SUNDIALS_DOUBLE_PRECISION)
782         printf("␣%lg",tempc[is]);
783   #else
784         printf("␣%g",tempc[is]);
785   #endif
786       }
787       printf("\n\n");
788     }
789   }
790
791   /*
792    * Print final statistics contained in iopt
793    */
794
795   static void PrintFinalStats(void *kmem)
796   {
797     long int nni, nfe, nli, npe, nps, ncfl, nfeSG;
798     int flag;
799
800     flag = KINGetNumNonlinSolvIters(kmem, &nni);
801     check_flag(&flag, "KINGetNumNonlinSolvIters", 1, 0);
802     flag = KINGetNumFuncEvals(kmem, &nfe);
803     check_flag(&flag, "KINGetNumFuncEvals", 1, 0);
804     flag = KINSpilsGetNumLinIters(kmem, &nli);
805     check_flag(&flag, "KINSpilsGetNumLinIters", 1, 0);
806     flag = KINSpilsGetNumPrecEvals(kmem, &npe);
807     check_flag(&flag, "KINSpilsGetNumPrecEvals", 1, 0);
808     flag = KINSpilsGetNumPrecSolves(kmem, &nps);
809     check_flag(&flag, "KINSpilsGetNumPrecSolves", 1, 0);
810     flag = KINSpilsGetNumConvFails(kmem, &ncfl);
811     check_flag(&flag, "KINSpilsGetNumConvFails", 1, 0);
812     flag = KINSpilsGetNumFuncEvals(kmem, &nfeSG);
813     check_flag(&flag, "KINSpilsGetNumFuncEvals", 1, 0);
814
815     printf("Final␣Statistics..␣\n");
816     printf("nni␣␣␣␣=␣%5ld␣␣␣␣nli␣␣␣=␣%5ld\n", nni, nli);
817     printf("nfe␣␣␣␣=␣%5ld␣␣␣␣nfeSG␣=␣%5ld\n", nfe, nfeSG);
818     printf("nps␣␣␣␣=␣%5ld␣␣␣␣npe␣␣␣=␣%5ld␣␣␣␣␣ncfl␣␣=␣%5ld\n", nps, npe, ncfl);
819
820   }
821
822   /*
823    * Routine to send boundary data to neighboring PEs
824    */
```

```
825
826 static void BSend(MPI_Comm comm, long int my_pe,
827                   long int isubx, long int isuby,
828                   long int dsizex, long int dsizey, realtype *cdata)
829 {
830   int i, ly;
831   long int offsetc, offsetbuf;
832   realtype bufleft[NUM_SPECIES*MYSUB], bufright[NUM_SPECIES*MYSUB];
833
834   /* If isuby > 0, send data from bottom x-line of u */
835   if (isuby != 0)
836     MPI_Send(&cdata[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);
837
838   /* If isuby < NPEY-1, send data from top x-line of u */
839   if (isuby != NPEY-1) {
840     offsetc = (MYSUB-1)*dsizex;
841     MPI_Send(&cdata[offsetc], dsizex, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
842   }
843
844   /* If isubx > 0, send data from left y-line of u (via bufleft) */
845   if (isubx != 0) {
846     for (ly = 0; ly < MYSUB; ly++) {
847       offsetbuf = ly*NUM_SPECIES;
848       offsetc = ly*dsizex;
849       for (i = 0; i < NUM_SPECIES; i++)
850         bufleft[offsetbuf+i] = cdata[offsetc+i];
851     }
852     MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
853   }
854
855   /* If isubx < NPEX-1, send data from right y-line of u (via bufright) */
856   if (isubx != NPEX-1) {
857     for (ly = 0; ly < MYSUB; ly++) {
858       offsetbuf = ly*NUM_SPECIES;
859       offsetc = offsetbuf*MXSUB + (MXSUB-1)*NUM_SPECIES;
860       for (i = 0; i < NUM_SPECIES; i++)
861         bufright[offsetbuf+i] = cdata[offsetc+i];
862     }
863     MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
864   }
865 }
866
867 /*
868  * Routine to start receiving boundary data from neighboring PEs.
869  *  Notes:
870  *  1) buffer should be able to hold 2*NUM_SPECIES*MYSUB realtype entries,
871  *     should be passed to both the BRecvPost and BRecvWait functions, and
872  *     should not be manipulated between the two calls.
873  *  2) request should have 4 entries, and should be passed in both calls also.
874  */
875
876 static void BRecvPost(MPI_Comm comm, MPI_Request request[], long int my_pe,
877                       long int isubx, long int isuby,
878                       long int dsizex, long int dsizey,
879                       realtype *cext, realtype *buffer)
880 {
881   long int offsetce;
882
883   /* Have bufleft and bufright use the same buffer */
```

```
884   realtype *bufleft = buffer, *bufright = buffer+NUM_SPECIES*MYSUB;
885
886   /* If isuby > 0, receive data for bottom x-line of cext */
887   if (isuby != 0)
888     MPI_Irecv(&cext[NUM_SPECIES], dsizex, PVEC_REAL_MPI_TYPE,
889               my_pe-NPEX, 0, comm, &request[0]);
890
891   /* If isuby < NPEY-1, receive data for top x-line of cext */
892   if (isuby != NPEY-1) {
893     offsetce = NUM_SPECIES*(1 + (MYSUB+1)*(MXSUB+2));
894     MPI_Irecv(&cext[offsetce], dsizex, PVEC_REAL_MPI_TYPE,
895               my_pe+NPEX, 0, comm, &request[1]);
896   }
897
898   /* If isubx > 0, receive data for left y-line of cext (via bufleft) */
899   if (isubx != 0) {
900     MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
901               my_pe-1, 0, comm, &request[2]);
902   }
903
904   /* If isubx < NPEX-1, receive data for right y-line of cext (via bufright) */
905   if (isubx != NPEX-1) {
906     MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
907               my_pe+1, 0, comm, &request[3]);
908   }
909 }
910
911 /*
912  * Routine to finish receiving boundary data from neighboring PEs.
913  *  Notes:
914  *  1) buffer should be able to hold 2*NUM_SPECIES*MYSUB realtype entries,
915  *     should be passed to both the BRecvPost and BRecvWait functions, and
916  *     should not be manipulated between the two calls.
917  *  2) request should have 4 entries, and should be passed in both calls also.
918  */
919
920 static void BRecvWait(MPI_Request request[], long int isubx,
921                       long int isuby, long int dsizex, realtype *cext,
922                       realtype *buffer)
923 {
924   int i, ly;
925   long int dsizex2, offsetce, offsetbuf;
926   realtype *bufleft = buffer, *bufright = buffer+NUM_SPECIES*MYSUB;
927   MPI_Status status;
928
929   dsizex2 = dsizex + 2*NUM_SPECIES;
930
931   /* If isuby > 0, receive data for bottom x-line of cext */
932   if (isuby != 0)
933     MPI_Wait(&request[0],&status);
934
935   /* If isuby < NPEY-1, receive data for top x-line of cext */
936   if (isuby != NPEY-1)
937     MPI_Wait(&request[1],&status);
938
939   /* If isubx > 0, receive data for left y-line of cext (via bufleft) */
940   if (isubx != 0) {
941     MPI_Wait(&request[2],&status);
942
```

```
943       /* Copy the buffer to cext */
944       for (ly = 0; ly < MYSUB; ly++) {
945         offsetbuf = ly*NUM_SPECIES;
946         offsetce = (ly+1)*dsizex2;
947         for (i = 0; i < NUM_SPECIES; i++)
948           cext[offsetce+i] = bufleft[offsetbuf+i];
949       }
950     }
951
952     /* If isubx < NPEX-1, receive data for right y-line of cext (via bufright) */
953     if (isubx != NPEX-1) {
954       MPI_Wait(&request[3],&status);
955
956       /* Copy the buffer to cext */
957       for (ly = 0; ly < MYSUB; ly++) {
958         offsetbuf = ly*NUM_SPECIES;
959         offsetce = (ly+2)*dsizex2 - NUM_SPECIES;
960         for (i = 0; i < NUM_SPECIES; i++)
961           cext[offsetce+i] = bufright[offsetbuf+i];
962       }
963     }
964   }
965   /*
966    * Check function return value...
967    *    opt == 0 means SUNDIALS function allocates memory so check if
968    *             returned NULL pointer
969    *    opt == 1 means SUNDIALS function returns a flag so check if
970    *             flag >= 0
971    *    opt == 2 means function allocates memory so check if returned
972    *             NULL pointer
973    */
974
975   static int check_flag(void *flagvalue, char *funcname, int opt, int id)
976   {
977     int *errflag;
978
979     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
980     if (opt == 0 && flagvalue == NULL) {
981       fprintf(stderr,
982               "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
983               id, funcname);
984       return(1);
985     }
986
987     /* Check if flag < 0 */
988     else if (opt == 1) {
989       errflag = (int *) flagvalue;
990       if (*errflag < 0) {
991         fprintf(stderr,
992                 "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
993                 id, funcname, *errflag);
994         return(1);
995       }
996     }
997
998     /* Check if function returned NULL pointer - no memory allocated */
999     else if (opt == 2 && flagvalue == NULL) {
1000      fprintf(stderr,
1001              "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
```

```
1002                id, funcname);
1003       return(1);
1004     }
1005
1006     return(0);
1007   }
```

D Listing of fkinkryx.f

```
      program fkinkryx
c     ----------------------------------------------------------------
c     $Revision: 1.1 $
c     $Date: 2006/07/05 15:50:10 $
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: fkinkryx
c     ----------------------------------------------------------------
c
      implicit none

      integer ier, globalstrat, maxl, maxlrst
      integer*4 PROBSIZE
      parameter(PROBSIZE=128)
      integer*4 neq, i, msbpre
      integer*4 iout(15)
      double precision pp, fnormtol, scsteptol
      double precision rout(2), uu(PROBSIZE), scale(PROBSIZE)
      double precision constr(PROBSIZE)

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

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

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

      call fnvinits(3, neq, ier)
      if (ier .ne. 0) then
         write(6,1220) ier
 1220    format('SUNDIALS_ERROR: FNVINITS returned IER = ', i2)
         stop
      endif

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

```
 20   continue

      call fkinmalloc(iout, rout, ier)
      if (ier .ne. 0) then
         write(6,1230) ier
1230     format('SUNDIALS_ERROR: FKINMALLOC returned IER = ', i2)
         stop
      endif

      call fkinsetiin('MAX_SETUPS', msbpre, ier)
      call fkinsetrin('FNORM_TOL', fnormtol, ier)
      call fkinsetrin('SSTEP_TOL', scsteptol, ier)
      call fkinsetvin('CONSTR_VEC', constr, ier)

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

      call fkinspilssetprec(1, ier)

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

      call fkinsol(uu, globalstrat, scale, scale, ier)
      if (ier .lt. 0) then
         write(6,1242) ier, iout(9)
1242     format('SUNDIALS_ERROR: FKINSOL returned IER = ', i2, /,
     1          '                  Linear Solver returned IER = ', i2)
         call fkinfree
         stop
      endif

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

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

      do 30 i = 1, neq, 4
         write(6,1256) i, uu(i), uu(i+1), uu(i+2), uu(i+3)
1256     format(i4, 4(1x, f10.6))
30    continue

      write(6,1267) iout(3), iout(14), iout(4), iout(12), iout(13),
     1              iout(15)
1267  format(//'Final statistics:'//
     1     '  nni = ', i3, ',  nli  = ', i3, /,
     2     '  nfe = ', i3, ',  npe  = ', i3, /,
     3     '  nps = ', i3, ',  ncfl = ', i3)

      call fkinfree
```

```
117
118         stop
119         end
120
121   c * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
122   c     The function defining the system f(u) = 0 must be defined by a Fortran
123   c     function of the following form.
124
125         subroutine fkfun(uu, fval, ier)
126
127         implicit none
128
129         integer ier
130         integer*4 neq, i
131         double precision fval(*), uu(*)
132
133         common /psize/ neq
134
135         do 10 i = 1, neq
136            fval(i) = uu(i) * uu(i) - i * i
137    10   continue
138
139         ier = 0
140
141         return
142         end
143
144
145   c * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
146   c     The routine kpreco is the preconditioner setup routine. It must have
147   c     that specific name be used in order that the c code can find and link
148   c     to it.  The argument list must also be as illustrated below:
149
150         subroutine fkpset(udata, uscale, fdata, fscale,
151        1                  vtemp1, vtemp2, ier)
152
153         implicit none
154
155         integer ier
156         integer*4 neq, i
157         double precision pp
158         double precision udata(*), uscale(*), fdata(*), fscale(*)
159         double precision vtemp1(*), vtemp2(*)
160
161         common /pcom/ pp(128)
162         common /psize/ neq
163
164         do 10 i = 1, neq
165            pp(i) = 0.5d0 / (udata(i) + 5.0d0)
166    10   continue
167         ier = 0
168
169         return
170         end
171
172
173   c * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
174   c     The routine kpsol is the preconditioner solve routine. It must have
175   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
```

E Listing of fkinkryx_p.f

```
      program fkinkryx_p
c     ----------------------------------------------------------------
c     $Revision: 1.1 $
c     $Date: 2006/07/05 15:50:10 $
c     ----------------------------------------------------------------
c     Programmer(s): Allan G. Taylor, Alan C. Hindmarsh and
c                    Radu Serban @ LLNL
c     ----------------------------------------------------------------
c     Simple diagonal test with Fortran interface, using
c     user-supplied preconditioner setup and solve routines (supplied
c     in Fortran, below).
c
c     This example does a basic test of the solver by solving the
c     system:
c              f(u) = 0  for
c              f(u) = u(i)^2 - i^2
c
c      No scaling is done.
c      An approximate diagonal preconditioner is used.
c
c      Execution command: mpirun -np 4 fkinkryx_p
c     ----------------------------------------------------------------
c
      implicit none

      include "mpif.h"

      integer ier, size, globalstrat, rank, mype, npes
      integer maxl, maxlrst
      integer*4 localsize
      parameter(localsize=32)
      integer*4 neq, nlocal, msbpre, baseadd, i, ii
      integer*4 iout(15)
      double precision rout(2)
      double precision pp, fnormtol, scsteptol
      double precision uu(localsize), scale(localsize)
      double precision constr(localsize)

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

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

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

      call mpi_init(ier)
```

```
58        if (ier .ne. 0) then
59           write(6,1210) ier
60     1210  format('MPI_ERROR: MPI_INIT returned IER = ', i2)
61           stop
62        endif
63
64        call fnvinitp(mpi_comm_world, 3, nlocal, neq, ier)
65        if (ier .ne. 0) then
66           write(6,1220) ier
67     1220  format('SUNDIALS_ERROR: FNVINITP returned IER = ', i2)
68           call mpi_finalize(ier)
69           stop
70        endif
71
72        call mpi_comm_size(mpi_comm_world, size, ier)
73        if (ier .ne. 0) then
74           write(6,1222) ier
75     1222  format('MPI_ERROR: MPI_COMM_SIZE returned IER = ', i2)
76           call mpi_abort(mpi_comm_world, 1, ier)
77           stop
78        endif
79
80        if (size .ne. 4) then
81           write(6,1230)
82     1230  format('MPI_ERROR: must use 4 processes')
83           call mpi_finalize(ier)
84           stop
85        endif
86        npes = size
87
88        call mpi_comm_rank(mpi_comm_world, rank, ier)
89        if (ier .ne. 0) then
90           write(6,1224) ier
91     1224  format('MPI_ERROR: MPI_COMM_RANK returned IER = ', i2)
92           call mpi_abort(mpi_comm_world, 1, ier)
93           stop
94        endif
95
96        mype = rank
97        baseadd = mype * nlocal
98
99        do 20 ii = 1, nlocal
100          i = ii + baseadd
101          uu(ii) = 2.0d0 * i
102          scale(ii) = 1.0d0
103          constr(ii) = 0.0d0
104    20    continue
105
106       call fkinmalloc(iout, rout, ier)
107
108       if (ier .ne. 0) then
109          write(6,1231)ier
110    1231  format('SUNDIALS_ERROR: FKINMALLOC returned IER = ', i2)
111          call mpi_abort(mpi_comm_world, 1, ier)
112          stop
113       endif
114
115       call fkinsetiin('MAX_SETUPS', msbpre, ier)
116       call fkinsetrin('FNORM_TOL', fnormtol, ier)
```

```
      call fkinsetrin('SSTEP_TOL', scsteptol, ier)
      call fkinsetvin('CONSTR_VEC', constr, ier)

      call fkinspgmr(maxl, maxlrst, ier)
      call fkinspilssetprec(1, ier)

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

      call fkinsol(uu, globalstrat, scale, scale, ier)
      if (ier .lt. 0) then
         write(6,1242) ier, iout(9)
 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) iout(3), iout(14), iout(4),
     1                               iout(12), iout(13), iout(15)
 1267 format(/'Final statistics:'//
     1     '  nni = ', i3, ',  nli  = ', i3, /,
     2     '  nfe = ', i3, ',  npe  = ', i3, /,
     3     '  nps = ', i3, ',  ncfl = ', i3)

      call fkinfree

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

      stop
      end


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

      subroutine fkfun(uu, fval, ier)

      implicit none

```

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

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

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

      return
      end


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

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

      implicit none

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

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

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

      ier = 0

      return
      end


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

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

      implicit none

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

```
235         double precision vv(*), ftem(*)
236         double precision pp
237
238         common /pcom/ pp(localsize), mype, npes, baseadd, nlocal
239
240         do 10 i = 1, nlocal
241    10      vv(i) = vv(i) * pp(i)
242
243         ier = 0
244
245         return
246         end
```