User Documentation for KINSOL,
A Nonlinear Solver for Sequential and Parallel Computers

A.G. Taylor
A.C. Hindmarsh

Center for
Applied Scientific Computing

UCRL-ID-131185
July 1998
DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from the Office of Scientific and Technical Information
P O Box 62, Oak Ridge, TN 37831
Prices available from (423) 576-8401

Available to the public from the National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd., Springfield, VA 22161
1. Introduction. KINSOL is a general purpose nonlinear system solver callable from either C or Fortran programs. It is based on NKSOL [3], but is written in ANSI-standard C rather than Fortran. Its most notable feature is that it uses Krylov Inexact Newton techniques in the system’s approximate solution, thus sharing significant modules previously written within CASC at LLNL to support CVODE[6, 7]/PVODE[9, 5]. It also requires almost no matrix storage for solving the Newton equations as compared to direct methods. The name KINSOL is derived from those techniques Krylov Inexact Newton SOLver. The package was arranged so that selecting one of two forms of a single module in the compilation process will allow the entire package to be created in either sequential (serial) or parallel form. The parallel version of KINSOL uses MPI (Message-Passing Interface) [8] and an appropriately revised version of the vector module NVECTOR, as mentioned above, to achieve parallelism and portability. KINSOL in parallel form is intended for the SPMD (Single Program Multiple Data) model with distributed memory, in which all vectors are identically distributed across processors. In particular, the vector module NVECTOR is designed to help the user assign a contiguous segment of a given vector to each of the processors for parallel computation. Several primitives were added to NVECTOR as originally written for PVODE to implement KINSOL.

KINSOL has been run on a Cray-T3D, an eight-processor DEC ALPHA and a cluster of workstations. It is currently being used in a simulation of tokamak edge plasmas and in groundwater two-phase flow studies at LLNL.

The remainder of this paper is organized as follows. Section 2 sets the mathematical notation and summarizes the basic methods. Section 3 summarizes the organization of the KINSOL solver, while Section 4 summarizes its usage. Section 5 describes a preconditioner module, Section 6 describes a set of Fortran/C interfaces, Section 7 describes an example problem, and Section 8 discusses availability.

2. Mathematical Considerations. The KINSOL code is a C implementation of a previous code, NKSOL, a nonlinear system solver written in Fortran by Brown and Saad [3].

The nonlinear system of equations

\[ F(u) = 0, \]

where \( F(u) \) is a nonlinear function from \( \mathbb{R}^N \) to \( \mathbb{R}^N \), is solved by this package. An Inexact Newton method is applied to (1) resulting in the following iteration.

---

* Research performed under the auspices of the U.S. Department of Energy, by Lawrence Livermore National Laboratory under contract W-7405-ENG-48. Work supported by LDRD, Project 95-ER-036

† Center for Applied Scientific Computing, L-561, LLNL, Livermore, CA 94551
Inexact Newton iteration
1. Set $u_0$ = an initial guess
2. For $n = 0, 1, 2$, until convergence do
   (a) Solve $J(u_n)\delta_n = -F(u_n)$
   (b) Set $u_{n+1} = u_n + \delta_n$
   (c) Test for convergence

Here, $J(u_n) = F'(u_n)$ is the system Jacobian. As this code module is anticipated for use on large systems, only iterative methods were considered to solve the system in step 2(a). These solutions are only approximate. Methods of this type used for solution of nonlinear systems are called Inexact Newton methods. At each stage in the iteration process, a multiple of the approximate solution $\delta_n$ to the equation of step 2(a) is applied to the previously determined iterated approximate solution to produce a new approximate solution. Convergence is tested before iteration continues. The iterative method currently implemented is one of the class of Krylov methods.

As only the matrix-vector product $J(u)v$ is required in the Krylov method, in this nonlinear equations setting that action is approximated by a difference quotient of the form

$$J(u)v \approx \frac{F(u + \sigma v) - F(u)}{\sigma},$$

where $u$ is the current approximation to a root of (1) and $\sigma$ is a scalar, appropriately chosen to minimize numerical error in the computation of (2). An optional user-defined routine implementing this matrix-vector product is accommodated. See further details below in the section describing the routine KINSpgmr.

To the above methods are added scaling and preconditioning. Scaling is allowed for both the approximate solution vector and the system function vector. Additionally, right preconditioning is provided for if the preconditioning setup and solve routines are supplied by the user.

While only one linear solver is now implemented for use with this package, the formal structure is in place for alternate solvers. The solver currently implemented is the GMRES solver [2, 10] in module SPGMR and accessed via KINSpgmr. Here GMRES stands for Generalized Minimal RESidual. In most cases, performance of SPGMR is improved by user-supplied preconditioners.

SPGMR is one of a class of preconditioned Krylov methods. Write the linear system simply as

$$Ax = b$$

A preconditioned Krylov method for (3) involves a preconditioner matrix $P$ that approximates $A$, but for which linear systems $Px = b$ can be solved easily. For preconditioning on the right, the Krylov method is applied to the transformed system

$$(AP^{-1})(Px) = b$$

In KINSOL, the user may precondition the system on the right or use no preconditioner. In any case, the Krylov method (in our case GMRES) is applied to the transformed system

$$\bar{A}\bar{x} = \bar{b}$$
From an initial guess $\bar{x}_0$, an approximate solution $\bar{x}_m = \bar{x}_0 + z$ is obtained for $m = 1, 2, \ldots$ (until convergence), with $z$ chosen from the Krylov subspace $K_m = \text{span}\{r_0, Ar_0, \ldots, A^{m-1}r_0\}$ of dimension $m$, where $r_0$ is the initial residual $\bar{b} - \bar{A}\bar{x}_0$. Each Krylov iteration requires one matrix-vector multiply operation $\bar{A}w$, which is a combination of multiplies by $\bar{A}$ and by $P^{-1}$. Multiplication of a given vector $v$ by $\bar{A}$ requires the product $\bar{J}v$, and that is approximated by a difference quotient $[F(u + \sigma v) - F(u)]/\sigma$. Multiplication by $P^{-1}$ is to be provided by the user of the solver, and is generally problem-dependent. In the case of GMRES, the choice in $K_m$ is based on minimizing the $L_2$ norm of the residual $\bar{b} - \bar{A}\bar{x}_m$ [2, 10]. When a given $\bar{x}_m$ meets the linear system convergence criterion, $\delta_n$ corresponds to the next increment $\delta_n$ in the solution of (1). $\delta_n$ is obtained from $\bar{x}_m$ by applying scaling and preconditioning. The increment $\delta_n$ is then added to $u_n$ to form $u_{n+1}$ in step 2(b) by one of the strategies discussed below. The new iterate $u_{n+1}$ is tested for (nonlinear) convergence in (1), which is step 2(c) of the Inexact Newton iteration.

Two methods of applying a computed step $\delta_n$ to the previously computed approximate solution vector are implemented. Denoted ‘global strategies’, they attempt to use the direction implied by $\delta_n$ in the most efficient way in furthering convergence of the global (i.e., nonlinear) problem. The first and simplest is the Inexact Newton strategy. A more advanced technique is implemented in the second strategy, called Linesearch. The so-called ‘Forcing Term’ algorithms of Eisenstat and Walker [4] to control the linear convergence tolerance are also implemented.

A fundamental set of mathematical operations on $N$-vectors has been written for both CVODE/PVODE and KINSOL. This set of computational kernels exists in a distinct code module called NVECTOR. By separating these frequent operations from the rest of the code, almost all operations in KINSOL with significant potential for parallel computation have been isolated. Then, two different sets of kernels, both with the same routine names and a common interface, allow parallel computation to be very simply implemented in these codes. The operations done by this set of kernels are vector addition, scaling, and copy, vector norms, scalar products, and so forth.

3. Code Organization. A way to visualize KINSOL is to think of the code as being organized in layers, as shown in Fig. 1. Here, a module’s name is used to indicate the general function of the module’s contents. Viewed this way, the user’s main program is at the top level. This program, with associated user-supplied routines, makes various initialization calls, manages input/output, and calls the KINSOL main module which carries out the system solution. At the next level down, the KINSOL main module controls the iterative solution process, and is independent of the linear system method. KINSOL calls the user-supplied function $F$, known as func internally, and accesses the linear system solver. At the third level is found the linear system solver KINSPGMR, which provides an interface to a generic solver for the SPGMR method, consisting of modules SPGMR and ITERATIV. KINSPGMR also accesses the user-supplied preconditioner solve routine psolve, if specified, and, if supplied, also accesses a user-supplied routine precondset that computes and preprocesses the preconditioner. The precondset routine is usually implemented by way of an approximate Jacobian matrix. Other linear system solvers may be added to the package in the future. Such additions will be independent of the KINSOL and KINSPGMR modules. Several supporting modules reside at
FIG 1 Overall structure of the KINSOL package. Modules comprising the central solver are distinguished by rounded boxes, while the user program, generic linear solvers, and auxiliary modules are in unrounded boxes.
The fourth level These include LLNLTYPES, LLNLMMATH, and NVECTOR. The first of these defines types real and integer. The second specifies power functions, and the third is discussed further below.

The key to being able to move from the sequential computing environment to the parallel computing environment lies in the NVECTOR module. This was briefly mentioned in the previous section. The idea is to distribute solution of the nonlinear system over several processors so that each processor is solving a contiguous subset of the system. This is achieved through the NVECTOR module, which handles all calculations on N-vector in a distributed manner, when the parallel version is compiled with parallel libraries. For any vector operation, each processor performs the operation on its contiguous elements of the input vectors, of length (say) Nlocal, followed by a global reduction operation where needed. In this way, vector calculations can be performed simultaneously with each processor working on its block of the vector. Vector kernels are designed to be used in a straightforward way for various vector operations that require the use of the entire distributed N-vector. These kernels include dot products, various norms, linear sums, and so on. The key to simply handling both parallel and serial applications of a code lies in standardizing the interface to the vector kernels both sequential and parallel versions of NVECTOR have an identical interface. In this way, one can access the kernels without referring directly to the underlying vector structure. This is assisted by using abstract data types that describe the machine environment data block (type machEnvType) and all N-vectors (type NVector). Functions to define a block of machine-dependent information and to free that block of information are also included in the vector module. Because the KINSOL interface to the vector kernels is independent of the vector structure, the user could supply their own kernel to best fit their application data structures. All references to parallelism are in the kernel, thus, the user would handle all parallel aspects in this case.

As the algorithms used in NKSOL had several unique features, notably the way that constraints were handled [3], several new vector kernels were written and added to the module NVECTOR. The changes, completely transparent to CVODE/PVODE, have now been incorporated in the 'common' version of NVECTOR.

The parallel version of KINSOL uses the MPI (Message Passing Interface) system [8] for all inter-processor communication. This achieves a high degree of portability, since MPI is becoming widely accepted as a standard for message passing software. For a different parallel computing environment, some rewriting of the vector module could allow the use of other specific machine-dependent instructions.

The coding style and structure of KINSOL was based on both style and structure of the preexisting CVODE/PVODE codes. This was predicated upon the requirement that the same vector kernel implementation and GMRES solvers be used in both codes. At the same time, those features somewhat unique to the Fortran language (e.g., those constructs used in the original code NKSOL), were placed appropriately in a C language setting. Considerable simplification of the calling sequences resulted from this process. Of course, the resulting C language structure maintains relative privacy for definitions for each portion of the code. The resulting code has proven to be readily adaptable to either sequential or parallel execution by means of two versions of the module NVECTOR.
4. Using KINSOL. This section is concerned with the use of KINSOL and consists of five subsections. Those subsections treat the user-callable routines constituting the KINSOL interface in an overview and then in detail, give a layout or skeleton of the user's main program, and user-supplied functions or routines, and discusses C++/C interfacing. The listing of the sample program KINXP (a Predator-Prey PDE problem, P is for parallel version) in the Appendix may be particularly helpful. That code is intended to serve as a template to assist in preparations to use KINSOL and is included in the KINSOL distribution package. The sequential equivalent of KINXP, called KINXS, and other variations and examples are found with KINSOL in the distribution package.

4.1. Overview of Routines and Their Usage. The source code is organized in files (modules) as shown in Table 1. For each module there are two corresponding files. For example, KINSOL requires both the files kinsol.c and kinsol.h.

<table>
<thead>
<tr>
<th>Module name</th>
<th>User-callable routines</th>
<th>other contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>KINSOL</td>
<td>KINMalloc, KINSol,</td>
<td>system function type SysFn, linear solver</td>
</tr>
<tr>
<td></td>
<td>KINFree</td>
<td>function pointers limit, lsetup, lsolve, lfree</td>
</tr>
<tr>
<td>KINSPGMR</td>
<td>KINSpgmr</td>
<td>KINSpgmrPrecondFn type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>KINSpgmrPrecondSolveFn type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>KINSpgmrAtimesFn type</td>
</tr>
<tr>
<td>SPGMR</td>
<td></td>
<td>SpgmrMalloc, SpgmrSolve, SpgmrFree</td>
</tr>
<tr>
<td>ITERATIVE</td>
<td></td>
<td>Routines in support of SPGMR</td>
</tr>
<tr>
<td>MVFCTOR</td>
<td>PVectInitMPI,</td>
<td>Type N.Vector; vector macros</td>
</tr>
<tr>
<td></td>
<td>PVectFreeMPI,</td>
<td>N_VMAKE, N_VDATA, etc</td>
</tr>
<tr>
<td></td>
<td>19 other vector kernels</td>
<td></td>
</tr>
<tr>
<td>LLNLMath</td>
<td>UnitRoundoff, RPowerI,</td>
<td>Macros MIN, MAX, ABS, SQR</td>
</tr>
<tr>
<td></td>
<td>RPowerR, RSqrt,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Macros MIN, MAX, ABS, SQR</td>
<td></td>
</tr>
<tr>
<td>LLNLTYPES</td>
<td>Types real, integer, bool</td>
<td></td>
</tr>
</tbody>
</table>

Table 1
Modules in the KINSOL package

In addition to routines supplied with KINSOL, there are several routines either required or optional that the user can supply. They are outlined in Table 2. Details and use of the last two routines listed there are discussed in Section 5.

4.2. Detailed description of routines. This subsection uses extracts from header files for KINSOL and KINSPGMR to detail the arguments of user-callable routines. For each routine, the declaration with arguments is followed by a section of comments. Please note that the system function $F(u)$ is called func(uu) in the actual KINSOL and KINSPGMR source code. The independent variable $u$ is called uu in those code modules as well.

4.2.1. Memory allocation routine KINMalloc.
void *KINMalloc(integer Neq, FILE *msgfp, void *machEnv);

/******************************************************************************
 * Function : KINMalloc
 * This function allocates main memory for the KINSol package. It also allocates several vectors of size Neq used by the package. Other N-Vectors are also to be allocated by the user and supplied to KINSol
 * Neq size of vectors being handled by the current memory
 * msgfp pointer to a FILE used to receive error messages from KINMalloc
 */
******************************************************************************

4.2.2. Main solver KINSol.

int KINSol(void *kinmem, integer Neq, 
N-Vector uu, SysFn func, int globalstrategy, 
N-Vector uscale, N-Vector fscale, 
real fnormtol, real scstepptol, N-Vector constraints, 
boole optIn, long int iopt[], real ropt[], void *f_data, 
FILE *msgfp, void *machEnv);

/******************************************************************************
 * Function : KINSol
 *-----------------------------------------------------------------
 * KINSol initializes memory for a problem previously allocated by
 */
******************************************************************************

<table>
<thead>
<tr>
<th>typedef name (* - optional)</th>
<th>purpose of user-supplied routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>SysFn</td>
<td>the function $F(u)$, also known as func(uu)</td>
</tr>
<tr>
<td>KINSpgmPrecondFn*</td>
<td>setup routine for preconditioner</td>
</tr>
<tr>
<td>KINSpgmPrecondSolveFn*</td>
<td>solve routine for preconditioner</td>
</tr>
<tr>
<td>KINSpgmuserAtimesFn*</td>
<td>user-supplied Atimes function</td>
</tr>
<tr>
<td>KINLocalFn*</td>
<td>local computation function (BBD preconditioner)</td>
</tr>
<tr>
<td>KINCommFn*</td>
<td>interprocessor communication function (BBD preconditioner)</td>
</tr>
</tbody>
</table>

**TABLE 2**
User-supplied routines for KINSOL
a call to KINMalloc. It also checks the initial value of uu (the initial guess) against the constraints and checks if the initial guess is a solution of the system. It then attempts to solve the system \( \text{func}(uu) = 0. \), where the function \( \text{func} \) is supplied by the user. The input arguments for KINSol and their function are described below:

- \( \text{Neq} \) is the number of equations in the algebraic system or, for a parallel problem, the number of variables assigned to the current processor.
- \( \text{kinmem} \) pointer to KINSol memory block returned by the preceding KINMalloc call.
- \( uu \) is the solution vector for the system \( \text{func}(uu) = 0. \). \( uu \) is to be set to an initial value if other vector starting value is desired.
- \( \text{func} \) is the system function for the system: \( \text{func}(uu) = 0. \)
- \( \text{globalstrategy} \) is a variable which indicates which global strategy to apply the computed increment delta in the solution \( uu \). Choices are:
  - INEXACT_NEWTON or LINESEARCH
- \( \text{uscale} \) is an array (type N_Vector) of diagonal elements of the scaling matrix for \( uu \). The elements of \( \text{uscale} \) must be positive values. The scaling matrix \( \text{uscale} \) should be chosen so that \( \text{uscale} \times uu \) (as a matrix multiplication) should have all its components with roughly the same magnitude when \( uu \) is close to a root of \( \text{func} \).
- \( \text{fscale} \) is an array (type N_Vector) of diagonal elements of the scaling matrix for \( \text{func} \). The elements of \( \text{fscale} \) must be positive values. The scaling matrix \( \text{fscale} \) should be chosen so that \( \text{fscale} \times \text{func}(uu) \) (as a matrix multiplication) should have all its components with roughly the same magnitude when \( uu \) is NOT too near a root of \( \text{func} \).
- \( \text{fnormtol} \) is a real (scalar) value containing the stopping tolerance on \( \maxnorm(\text{fscale} \times \text{func}(uu)) \).
  - If \( \text{fnormtol} \) is input as 0., then a default value of (uaround) to the 1/3 power will be used.
  - uaround is the unit roundoff for the machine in use for the calculation. (see UnitRoundoff in llnlmath module)
scsteptol is a real (scalar) value containing the stopping tolerance on the maximum scaled step
\( uu(k) - uu(k-1) \). If scsteptol is input as 0., then a default value of
to the 2/3 power will be used.

uround is the unit roundoff for the machine in use for the calculation. (see UnitRoundoff in
1lnlmath module)

constraints is a pointer to an array (type N_Vector) of

constraints on uu. If the pointer passed in is NULL, then NO constraints are applied to uu. A NULL pointer
also stops application of the constraint on the max relative change in uu, controlled by the input
variable relu which is input via ropt[RELU]
a positive value in constraints[i] implies that the ith component of uu is to be
constrained > 0
A negative value in constraints[i] implies that the ith component of uu is to be constrained < 0.
A zero value in constraints[i] implies there is no constraint on uu[i].

optIn is a flag (boole) indicating whether optional inputs from the user in the arrays iopt and ropt are to be
used. Pass FALSE to ignore all optional inputs and TRUE to use all optional inputs that are present.
Either choice does NOT affect outputs in other positions of iopt or ropt.

iopt is the user-allocated array (of size OPT_SIZE) that will hold optional integer inputs and outputs.
The user can pass NULL if he/she does not wish to use optional integer inputs or outputs.
If optIn is TRUE, the user should preset to 0 those locations for which default values are to be used.
Elements of iopt which have significance for either input or output parameters are:
PRINTFL, MXITER, PRECOND_NO_INIT, NNI ,NFE ,NBCF,
NBKTRK, MXKRYL, and ETACHoice

ropt is the user-allocated array (of size OPT_SIZE) that will hold optional real inputs and outputs.
The user can pass NULL if he/she does not wish to use optional real inputs or outputs.
If optIn is TRUE, the user should preset to 0 0 the optional input locations for which default values are
to be used.
Elements of iopt which have significance for either input or output parameters are:
* MXNEWTSTEP, RELFUNC, RELU, FNORM, STEPL, ETACONST, *
* ETAGAMMA, and ETAALPHA
*
* Permissible iopt and ropt input parameters are given *
* in a section below.
*
* f_data is a pointer to work space for use by the user-supplied*
* function func. The space allocated to f_data is *
* allocated by the user's program before the call to *
* KINMalloc.
*
* msgfp is the file pointer for a message file where all KINSol*
* warning, error and informational messages will be *
* written. This parameter can be stdout (standard output)*
* , stderr (standard error), a file pointer to a user *
* created file, or NULL. If NULL is passed, then stdout *
* (standard output) is used as a default.
*
* machEnv is a pointer to machine environment-specific *
* information. Pass NULL for the sequential case *
* (see nvector.h)
*
* If successful, KINMalloc returns a pointer to initialized *
* problem memory. This pointer should be passed to KINSol If *
* an initialization error occurs, KINMalloc prints an error *
* message to the file specified by msgfp and returns NULL.
*
******************************************************************

4.2.3. Main solver KINSol optional inputs and outputs. The input of several optional input parameters is handled by placing their values in appropriate elements of either iopt or ropt arrays. Those optional input parameters and their permissible input values are now discussed.

*******************************************************************************
* Optional Inputs and Outputs
*******************************************************************************
* The user should declare two arrays for optional input and *
* output, an iopt array for optional integer input and output *
* and an ropt array for optional real input and output. These *
* arrays should both be of size OPT_SIZE.
* So the user's declaration should look like:
* *
* long int iopt[opt_SIZE];
* real ropt[opt_SIZE];
The following definitions are indices into the iopt and ropt arrays. A brief description of the contents of these positions follows:

**iopt[PRINTFL]** (input) allows user to select from 4 levels of output to FILE msgfp.
- **=0** no statistics printed (DEFAULT)
- **=1** output the nonlinear iteration count, the scaled norm of \( f(u) \), and number of func calls.
- **=2** same as 1 with the addition of global strategy statistics:
  - \( f_1 = 0.5 \times \| f \|_2^2 \) and \( f_1new = 0.5 \times \| f \|_2^2 \).
- **=3** same as 2 with the addition of further Krylov iteration statistics.

**iopt[MXITER]** (input) maximum allowable number of nonlinear iterations. The default is MXITER_DEFAULT.

**iopt[PRECOND_NO_INIT]** (input) Set to 1 to prevent the initial call to the routine precondset upon a given call to KINSol. Set to 0 or leave unset to force the initial call to precondset. Use the choice of 1 only after beginning the first of a series of calls with a 0 value. If a value other than 0 or 1 is encountered, the default, 0, is set in this element of iopt and thus the routine precondset will be called upon every call to KINSol, unless iopt[PRECOND_NO_INIT] is changed by the user.

**iopt[ETACHOICE]** (input) a flag indicating which of three methods to use for computing \( \eta \), the coefficient in the linear solver convergence tolerance \( \epsilon \) given by
- \( \epsilon = (\eta + u_{\text{round}}) \times \| f(u) \|_2 \)
  - here, all norms are the scaled L2 norm.
  - The linear solver attempts to produce a step \( p \) such that \( \| f(u) + J(u) \cdot p \| \leq \epsilon \).
  - Two of the methods for computing \( \eta \) calculate a value based on the convergence process in the routine KINForcingTerm.
  - The third method does not require calculation; a constant \( \eta \) is selected.

The default if iopt[ETACHOICE] is not
specified is ETACHOICE1, (see below)

The allowed values (methods) are:

ETACONSTANT constant eta, default of 0.1 or user supplied choice, for which see ropt[ETACONST],

\[ \eta(k) = \frac{\text{ABS}( \| \text{func}(uu(k)) \| - \| \text{func}(uu(k-i)) + J(uu(k-1)) \| p )}{\| \text{func}(uu(k-i)) \|} \]

ETACHOICE2 which uses choice 2 of Eisenstat and Walker wherein eta is:
\[ \eta(k) = \eta_{\text{safe}} \]
\[ \eta_{\text{safe}} = \eta_{\text{safe}} \]

eta(k) = egamma * 
\[ \left( \frac{\| \text{func}(uu(k)) \|}{\| \text{func}(u(k-1)) \|} \right)^{ealpha} \]

egamma and ealpha for choice 2, both required, are from either defaults (egamma = 0.9 , ealpha = 2) or from user input, see ropt[ETALPHA] and ropt[ETAGAMMA], below.

For eta(k) determined by either Choice 1 or Choice 2, a value eta_safe is determined, and the safeguard \[ \eta_{\text{safe}} = \max(\eta_{\text{safe}}, \eta(k)) \]

is applied to prevent eta(k) from becoming too small to quickly.

For Choice 1, \[ \eta_{\text{safe}} = \eta(k-i)^{((l.+sqrt(5.))/2 )} \]

and for Choice 2, \[ \eta_{\text{safe}} = \eta_{\text{safe}} \]

(These safeguards are turned off if they drop below 0.1 Also, eta is never allowed to be less than \eta_{\text{min}} = 1.e-4 .)

iopt[NNI] (output) total number of nonlinear iterations

iopt[NFE] (output) total number of calls to the user-supplied system function func.

iopt[NBCF] (output) total number of times the beta condition could not be met in the linesearch algorithm. The nonlinear iteration is halted if this value ever exceeds MXNBCF (10)

iopt[NBKTRK] (output) total number of backtracks in the
linesearch algorithm

* ropt[MXNEWSTEP] (input) maximum allowable length of a newton step. The default value is calculated from
1000*max(norm(uscale*uu(0),norm(uscale))

* ropt[RELFUNC] (input) relative error in computing func(uu)
if known. Default is the machine epsilon

* ropt[RELU] (input) a scalar constraint which restricts the update of uu to del(uu)/uu < ropt[RELU]
The default is no constraint on the relative step in uu.

* ropt[ETAGAMMA] (input) the coefficient egamma in the eta computation. See routine KINForcingTerm
(SEE iopt[ETACHOICE] above for additional info)

* ropt[ETAALPHA] (input) the coefficient ealpha in the eta computation. See routine KINForcingTerm
(SEE iopt[ETACHOICE] above for additional info)

* ropt[ETACONST] (input) a user specified constant value for eta, used in lieu of that computed by routine KINForcingTerm
(SEE iopt[ETACHOICE] above for additional info)
Permissible ETACHOICE values are ETACHOICE1 (the default), ETACHOICE2, and ETACONST

* ropt[FNORM] (output) the scaled norm at a given iteration:
norm(fscale(func(uu))

* ropt[STEPL] (output) last step length in the global strategy routine:
KINLineSearch or KINInexactNewton)

4.2.4. Main solver (KINSol) return codes. The return code values for the routine KINSol, both for success and a variety of possible failures, are given next

* KINSol returns an integer-valued termination code with the set of possible values:
KINSOL_NO_MEM,KINSOL_INPUT_ERROR,
KINSOL_SUCCESS, KINSOL_SCALE_LT_FNORM,
KINSOL_LNSRCH_NONCONV, KINSOL_MAXITER_REACHED,
KINSOL_MXNEWT_5X_EXCEEDED, KINSOL_LINEARSEARCH_BCFAIL,
KINSOL_KRYLOV_FAILURE, KINSOL_PRECONDSET_FAILURE,  
KINSOL_PRECONDSOLVE_FAILURE,  
KINSOL_INITIAL_GUESS_OK  

The meanings of these return codes are now given, each by  
the suffix portion of the respective code. That is,  
KINSOL_NO_MEM is listed in the descriptions below as NO_MEM  

SUCCESS: means maxnorm(fscale*func(uu) <= fnormtol, where  
maxnorm() is the maximum norm function N_VMaxNorm  
uu is probably an approximate root of func.  

SCALED_LT_FNORM: means the scaled distance between the last  
two steps is less than scsteptol. uu may be an  
approximate root of func, but it is also possible  
that the algorithm is making very slow progress  
and is not near a root or that scsteptol is too  
large  

LNSRCH_NONCONV: means the LineSearch module failed to reduce  
norm(func) sufficiently on the last global step  
Either uu is close to a root of f and no more  
accuracy is possible, or the finite-difference  
approximation to j*v is inaccurate, or scsteptol  
is too large. Check the outputs ncfl and nni: if  
ncfl is close to nni, it may be the case that the  
Krylov iteration is converging very slowly. In  
this case, the user may want to use precondition- 
ing and/or increase the maxl value in the  
KINSpgmr input list (that is, increase the max  

dimension of the Krylov subspace by setting maxl  
to nonzero (thus not using the default value of  
KINSPGMR_MAXL), or if maxl is being set, increase  
it its value  

MAXITER_REACHED: means that the maximum allowable number of  
nonlinear iterations has been reached. This is by  
default 200, but may be changed through optional  
input iopt[MXITER].  

MXNEWT_5X_EXCEEDED: means 5 consecutive steps of length mxnewt*  
(maximum Newton stepsize limit) have been taken.  
Either norm(f) asymptotes from above to a finite  
value in some direction, or mxnewt is too small.  

Mxnewt is computed internally (by default) as  
mxnewt = 1000*max(nmax(uscale*uu0),1), where  

uu0 is the initial guess for uu, and norm() is
the Euclidean norm \( \| \cdot \|_{\text{rmsNorm}} \). \text{Mxnewt} can be set by the user through optional input \text{ropt}[\text{MXNEWTSTEP}].

\text{LINESEARCH_BCFAIL}: means that more than the allowed maximum number of failures (\text{MXNBCF}) occurred when trying to satisfy the beta condition in the linesearch algorithm. It is likely that the iteration is making poor progress.

\text{KRYLOV_FAILURE}: means there was a failure of the Krylov iteration process to converge.

\text{PRECONDSET_FAILURE}: means there was a nonrecoverable error in PrecondSet causing the iteration to halt.

\text{PRECONDSOLVE_FAILURE}: means there was a nonrecoverable error in PrecondSolve causing the iteration to halt.

\text{NO_MEM}: the KINSol memory pointer received was NULL.

\text{INPUT_ERROR}: one or more input parameters or arrays was in error. See the listing in msgfp for further info.

4.2.5. Deallocation routine \text{KINFree}. The next material describes the routine \text{KINFree}.

Note that it need not be called after a specific KINSol call but only when the memory used by the KINSOL package is to be released.

```c
void KINFree(void *kin_mem);
```

4.2.6. Linear solver interface function definitions. The linear solver package to be used with KINSOL interfaces with it via four routines of the type given below. Note that at present there are only the four routines (\text{KINSpgmrInit}, \text{KINSpgmrSetup}, \text{KINSpgmrSolve}, and \text{KINSpgmrFree}) from the KINSPGMR package available. In the following, each routine is named, followed by the generic description. If a user wishes to implement another linear
solver within KINSOL, the calling conventions given below need to be followed as well as
the entire interface as used in KINSPGMR

KINSpgmrInit:

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

KINSpgmrSetup:

/*****************************************************************************/
* *
* int (*kin_lsetup)(KINMem kin_mem);
* *-------------------------------------------------------------
* * The job of kin_lsetup is to prepare the linear solver for *
* subsequent calls to kin_lsolve *
* *
* kin_mem - problem memory pointer of type KINMem. See the big *
* typedef earlier in this file.
* *
* The kin_lsetup routine should return 0 if successful, *
* a positive value for a recoverable error, and a negative value *
* for an unrecoverable error.
* *
/*****************************************************************************/

KINSpgmrSolve:

/*****************************************************************************/
* *
* int (*kin_lsolve)(KINMem kin_mem, N_Vector bb, N_Vector xx, *
* real *res_norm); *
kin_lsolve must solve the linear equation $J x = b$, where $J$ is an approximate Jacobian matrix, $x$ is the approximate system solution, and the RHS vector $b$ is input. The solution is to be returned in the vector $b$. kin_lsolve returns a positive value for a recoverable error and a negative value for an unrecoverable error. Success is indicated by a 0 return value.

KINSpgmrFree:

```c
void (*kin_lfree)(KINMem kin_mem);
```

KINSpgmrFree should free up any memory allocated by the linear solver. This routine is called once a problem has been completed and the linear solver is no longer needed.

4.2.7. Linear solver routine KINSpgmr and its optional outputs. Pointers to the routines just described for the linear solver KINSpgmr are 'set' in the KINSOL memory structure by the call to KINSpgmr. No other action to prepare for those routines is required. KINSpgmr is now described.

```c
void KINSpgmr(void *kin_mem, int maxi, int maxlrst, int msbpre,
               KINSpgmrPrecondFn precondset,
               KINSpgmrPrecondSolveFn precondsolve,
               KINSpgmruserAtimesFn userAtimes,
               void *P_data);
```

Function : KINSpgmr

A call to the KINSpgmr function links the main KINSol solver with the KINSpgmr linear solver. Among other things, it sets the generic names linit, lsetup, lsolve, and lfree to the specific names for this package:

```c
   KINSpgmrInit
   KINSpgmrSetup
   KINSpgmrSolve
   KINSpgmrFree
```

kin_mem is the pointer to KINSol memory returned by
* KINSolMalloc.
* maxl is the maximum Krylov dimension. This is an
  optional input to the KINSpgmr solver. Pass 0 to
  use the default value MIN(Neq, KINSPGMR_MAXL=10)
* maxlrst is the maximum number of linear solver restarts
  allowed. Values outside the range 0 to 2*Neq/maxl
  will be restricted to that range. 0, meaning no
  restarts is a safe starting value.
* msbpre is the maximum number of steps calling the solver
  precondsolve without calling the preconditioner
  precondset. (The default is KINSPGMR_MSBPRE = 10)
* precondset is the user's preconditioner routine. It is used to
  evaluate and preprocess any Jacobian-related data
  needed by the precondsolve routine. See the
  documentation for the type KINSpgmrPrecondFn for
  full details. Pass NULL if no such setup of
  Jacobian data is required. A precondition is
  NOT required, but rather provided when needed by
  user's precondsolve routine
* precondsolve is the user's preconditioner solve routine. It
  is used to solve Px=b, where P is a preconditioner
  matrix. See the documentation for the type
  KINSpgmrPrecondSolveFn for full details. The only
  case in which precondset is allowed to be NULL is when
  no preconditioning is to be done. The NULL is taken
  as a flag that preconditioning is not desired.
* userAtimes is an optional routine supplied by the user to
  perform the matrix-vector multiply J v, where J is
  an approximate Jacobian matrix for that iteration.
  Enter NULL if no such routine is required. If one
  is supplied, conforming to the definitions given
  in this file, enter its filename.
* P_data is a pointer to user preconditioner data. This
  pointer is passed to precondset and precondsolve
  every time these routines are called.

Four elements in the KINSOL array iopt are used to return KINSPGMR statistics. Those iopt
elements are indexed by constants SPGMR_NLI, SPGMR_NPE, SPGMR_NPS, and SPGMR_NCFL, which are
declared in file kinspgmr.h. The meaning of each output parameter available for KINSpgmr is
explained next

/*******************************************************************************
* * KINSpgmr solver statistics indices *
* ******************************************************************************
* The following enumeration gives a symbolic name to each KINSpgmr-specific statistic
* The symbolic names are used as indices into the iopt and ropt arrays and values of both arrays
* are set in this module
* The KINSpgmr statistics are:
* + iopt[SPGMR-NLI] (output) number of linear iterations
* + iopt[SPGMR-NPE] (output) number of preconditioner evaluations
* + iopt[SPGMR-NPS] (output) number of calls made to user's psolve function.
* + iopt[SPGMR-NCFL] (output) number of linear convergence failures
*******************************************************************************

4.3. A Skeleton of the User's Main Program. The user's program must have the following steps in the order indicated

1. MPI_Init(&argc, &argv); to initialize MPI if used by the user's program Here argc and argv are the command line argument counter and array received by main
2. Set n, the local vector length (the sub-vector length for this processor), Neq, the global vector length (the problem size N, and the sum of all the values of Nlocal), and the active set of processors
3. machEnv = PVecInitMPI(comm, n, Neq, &argc, &argv), to initialize the NVECTOR module Here comm is the MPI communicator, which may be set in one of two ways If a proper subset of active processors is to be used, comm must be set by suitable MPI calls Otherwise, to specify that all processors are to be used, comm must be either MPI_COMM_WORLD or NULL.
4. Set the vector u of initial values Use the macro N_VMAKE(u, udata, machEnv), if an existing array udata contains the initial values of u Otherwise, make the call u = N_VNew(Neq, machEnv); and load initial values into the array defined by N_VDATA(u)
5. kmem = KINMalloc(. .), which allocates internal memory for KINSOL and returns a pointer to the KINSOL memory structure
6. KINSpgmr( .),
7. ier = KINSol(kmem, u, .), performs the solve
8. N_VDISPOSE, or N_VFree, upon completion of the integration, to deallocate the
memory for the vector u, allocated by \texttt{N\_VMAKE} or \texttt{N\_VNew}, respectively
9 \texttt{KINFree(kmem)}; to free the memory allocated for KINSOL
10 \texttt{PVecFreeMPI(machEnv)}; to free machine-dependent data

A summary of these in practice, for both the serial and parallel case, is given next

Summary of Serial Usage of KINSOL

1 \texttt{msgfile=fopen("*** out","w")},
2 Allocate and initialize vectors and structures, as required
3 \texttt{kmem= KINMalloc(SystemSize, msgfile, NULL)},
4 \texttt{KINSpgmr(kmem, ..)},
5 \texttt{retcode=KINSol(kmem, ..)},
6 \texttt{KINFree(mem)}

Summary of Parallel Usage of KINSOL

1 \texttt{msgfile = fopen("test.out", "w")}, Open message file, if desired
2 \texttt{MPI-Init()} as \texttt{PVecInitMPI}, below, also calls MPI-Init, this call is only required if the user’s program uses MPI before step 3
3 Set local length n and global length Neq, and the active set of processors
4 \texttt{machEnv = PVecInitMPI(comm, n, Neq, argc, argv)}; \texttt{comm = MPI communicator (if set up by user), or comm = MPI\_COMM\_WORLD or NULL (specifying all processors)} if \texttt{(machEnv == NULL)} return(1);
5 \texttt{N\_VMAKE(u, udata, machEnv)}; or \texttt{u = N\_VNew(Neq,machEnv)}, user sets up vectors, structures, etc
6 \texttt{kmem = KINMalloc(Neq, msgfile, machEnv)}, initializes KINSOL if stdout is to be used instead of a specific error message file, enter NULL in place of msgfile
7 \texttt{KINSpgmr( ..)}, call the setup routine for the linear solver to be used Note that only KINSpgmr is available at present
8 \texttt{flag= KINSol(kmem, Neq, u, func, .., machEnv)}; call the KINSOL main routine – can be called repetitively with different functions \texttt{func} and other options A linear solver choice made in step 7, when another choice is available, cannot be changed between KINSol calls
9 \texttt{N\_VDISPOSE( ..)}, or \texttt{N\_VFree( ..)}, call, as appropriate
10 \texttt{KINFree(kmem)}, Free KINSOL memory, independent of machine
11 \texttt{PVecFreeMPI(machEnv)}, Free machine-dependent data.

Every usage of KINSOL requires at least the inclusion of the following header files \texttt{kinsol.h}, \texttt{kinspgmr.h} or a future alternate solver, \texttt{math.h}, \texttt{llntypes.h}, and \texttt{nvector.h} If the BBD preconditioner is used, additional header files are required \texttt{kinbbdpre.h} and \texttt{band.h} The header file \texttt{mpi.h} is required for parallel applications of KINSOL
4.4. User-Supplied Functions. The function defining the nonlinear system, called \( F(u) \) in this report, but \( \text{func}(u) \) in KINSOL and KINSPGMR internal usage, must be of the form described by the following typedef extracted from KINSOL

```c
typedef void (*SysFn)(integer Neq, N_Vector uu, N_Vector fval, void *f_data);
```

Preconditioning is an important step in using KINSOL with any linear solver. The interface for the routines defining the preconditioner setup and solve routines for KINSPGMR are given next

```c
typedef int (*KINSpgmrPrecondFn)(integer Neq, N_Vector uu, N_Vector uscale, N_Vector fval, N_Vector fscale, N_Vector vtemp1, N_Vector vtemp2, SysFn func, real wound, long int *nfePtr, void *P_data);
```

The user-supplied preconditioner setup function `precondset` and the user-supplied preconditioner solve function `precondsolve` together must define the right preconditioner matrix \( P \) chosen so as to provide an easier system for the Krylov solver to solve. `precondset` is called to provide any matrix data required by the subsequent call(s) to `precondsolve`. The data is stored in the memory allocated to `P_data` and the structuring of that memory is up to the user. More specifically, the user-supplied preconditioner setup function `precondset` is to evaluate and preprocess any Jacobian-related data.
needed by the preconditioner solve function precondsolve. This might include forming a crude approximate Jacobian, and performing an LU factorization on the resulting approximation to J. This function will not be called in advance of every call to precondsolve, but instead will be called only as often as necessary to achieve convergence within the Newton iteration in KINSol. If the precondsolve function needs no preparation, the precondset function can be NULL.

precondset should not modify the contents of the arrays uu or fval as those arrays are used elsewhere in the iteration process.

Each call to the precondset function is preceded by a call to the system function func. Thus the precondset function can use any auxiliary data that is computed by the func function and saved in a way accessible to precondset.

The two scaling arrays, fscale and uscale, and unit roundoff uround are provided to the precondset function for possible use in approximating Jacobian data, e.g. by difference quotients. These arrays should also not be altered.

A function precondset must have the prototype given below. Its parameters are as follows:

Neq is the length of all vector arguments.

uu an N_Vector giving the current iterate for the system.

uscale an N_Vector giving the diagonal entries of the uu-scaling matrix.

fval an N_Vector giving the current function value

fscale an N_Vector giving the diagonal entries of the func-scaling matrix.

vtemp1 an N_Vector temporary

vtemp2 an N_Vector temporary

func the function func defines the system being solved: func(uu) = 0 , and its name is passed initially to KINSol in the call to KINMalloc

uround is the machine unit roundoff.
typedef int (*KINSpgmrPrecondSolveFn)(integer Neq,
        N_Vector uu, N_Vector uscale,
        N_Vector fval, N_Vector fscale,
        N_Vector vtem, N_Vector ftem,
        SysFn func, real u_round,
        long int *nfePtr, void *P-data);

/* Type : KINSpgmrPrecondSolveFn
   * The user-supplied preconditioner solve function precondsolve
   * is to solve a linear system P x = r in which the matrix P is
   * the (right) preconditioner matrix P
   *
   * precondset should not modify the contents of the iterate
   * array uu or the current function value array fval as those
   * are used elsewhere in the iteration process
   *
   * A function precondsolve must have the prototype given below.
   * Its parameters are as follows:
   * *
   * Neq     is the length of all vector arguments.
   * *
   * uu      an N_Vector giving the current iterate for the system. *
* uscale  an N_Vector giving the diagonal entries of the uu-
  scaling matrix

* fval  an N_Vector giving the current function value

* fscale  an N_Vector giving the diagonal entries of the func-
  scaling matrix

* vtem  an N_Vector work array, holds the RHS vector on input
  and the result x on output/return

* ftem  an N_Vector work array, usually set on input as vtemp

* func  the function func defines the system being solved:
  func(uu) = 0.

* uround  is the machine unit roundoff.

* nfePtr is a pointer to the memory location containing the
  KINSol problem data nfe = number of calls to func. The
  precondsolve routine should update this counter by
  adding on the number of func calls made in order to
  carry out the solution, if any. For example, if the
  routine calls func a total of W times, then the update
  is *nfePtr += W

* P_data is a pointer to user data - the same as the P_data
  parameter passed to KINSpgmr.

* Returned value:
  The value to be returned by the precondsolve function is a flag
  indicating whether it was successful. This value should be
  0 if successful,
  1 if failure, in which case KINSol stops

******************************************************************************/

The matrix-vector multiply $Ju$ may be done more efficiently on occasion by an algorithm
supplied by the user. This option is handled by supplying a routine of type next described
to KINSPGMR, the routine KINSpgmr, in particular

typedef int (*KINSpgmusruserAtimesFn)(void *f_data, N_Vector v,
N_Vector z, bool *new_uu,
N_Vector uu):

/******************************************************************************/

24
4.5. Use by a C++ Application. KINSOL has been written in so that it permits use by applications written in C++ as well as in C. For this purpose, each KINSOL header file is wrapped with conditionally compiled lines reading `extern "C" { . }`, conditional on the variable `_cplusplus` being defined. This directive causes the C++ compiler to use C-style names when compiling the function prototypes encountered. Users with C++ applications should also be aware that we have defined, in `llnltyps.h`, a boolean variable `type`, `boole`, since C has no such type. The type `boole` is equated to type `int`, and so arguments in user calls, or calls to user-supplied routines, which are of type `boole` can be typed as either `boole` or `int` by the user. The same applies to vector kernels which have a `type boole` return value, if the user is providing these kernels.

5. A Band-Block-Diagonal Preconditioner Module. A principal reason for using a parallel nonlinear system solver such as KINSOL lies in the solution of nonlinear systems arising in a partial differential equations (PDE) context. Moreover, the use of a Krylov iterative method for the solution of many such problems is motivated by the nature of the underlying linear system of equations that must be solved at each time step. The linear algebraic system is large, sparse, and structured. However, if a Krylov iterative method is to be effective in this setting, then an effective preconditioner needs to be used. Otherwise, the rate of convergence of the Krylov iterative method is usually unacceptably slow. Unfortunately, an effective preconditioner tends to be problem-specific.

However, we have developed one type of preconditioner that treats a rather broad class of problems. It has been successfully used for several realistic, large-scale problems and...
is included in a software module within the KINSOL package. This module generates a
preconditioner that is a block-diagonal matrix with each block being a band matrix. The
blocks need not have the same number of super- and sub-diagonals and these numbers
may vary from block to block. This Band-Block-Diagonal Preconditioner module is called
KINBBBDPre.

One way to envision these preconditioners is to think of the domain of the computa-
tional PDE problem as being subdivided into $M$ non-overlapping subdomains. Each of these
subdomains is then assigned to one of the $M$ processors to be used to solve the PDE system. The
basic idea is to isolate the preconditioning so that it is local to each processor, and also
to use a (possibly cheaper) approximate system function. This requires the definition of a
new function $g(u)$ which approximates the function $F(u)$ in the definition of the nonlinear
system (1). However, the user may set $g = F$. Corresponding to the domain decomposition,
there is a decomposition of the solution vector $u$ into $M$ disjoint blocks $u_m$, and a decompo-
sition of $g$ into blocks $g_m$. The block $g_m$ depends on $u_m$ and also on components of blocks
$u_{m'}$ associated with neighboring subdomains (so-called ghost-cell data). Let $\bar{u}_m$ denote $u_m$
augmented with those other components on which $g_m$ depends. Then we have

$$g(u) = [g_1(\bar{u}_1), g_2(\bar{u}_2), \ldots, g_M(t, \bar{u}_M)]^T$$

and each of the blocks $g_m(t, \bar{u}_m)$ is uncoupled from the others.

The preconditioner associated with this decomposition has the form

$$(5) \quad P = \text{diag}[P_1, P_2, \ldots, P_M]$$

where

$$(6) \quad P_m \approx J_m$$

and $J_m$ is a difference quotient approximation to $\partial g_m/\partial u_m$. This matrix is taken to be
banded, with upper and lower half-bandwidths $\mu_u$ and $\mu_l$ defined as the number of non-
zero diagonals above and below the main diagonal, respectively. The difference quotient
approximation is computed using $\mu_u + \mu_l + 2$ evaluations of $g_m$. The parameters $\mu_l$ and
$\mu_u$ need not be the true half-bandwidths of the Jacobian of the local block of $g$, if smaller
values provide a more efficient preconditioner. Also, they need not be the same on every
processor. The solution of the complete linear system

$$(7) \quad Px = b$$

reduces to solving each of the equations

$$(8) \quad P_m x_m = b_m$$

and this is done by banded LU factorization of $P_m$ followed by a banded backsolve.

To use this KINBBBDPre module, the user must supply two functions which the module
calls to construct $P$. These are in addition to the user-supplied system function $\text{func}$.
A function `glocfn(Nlocal, ulocal, glocal, f_data)` must be supplied by the user to compute \( g(u) \). It loads the real array `glocal` as a function of `t` and `ulocal`. Both `glocal` and `ulocal` are of length `Nlocal`, the local vector length.

A function `gcomm(Nlocal, u, f_data)` which must be supplied to perform all interprocessor communications necessary for the execution of the `glocfn` function, using the input vector `u` of type `N_Vector`.

Both functions take as input the same pointer `f_data` as that passed by the user to `KINMalloc` and passed to the user’s function `func`, and neither function has a return value. The user is responsible for providing space (presumably within `f_data`) for components of `u` that are communicated by `gcomm` from the other processors, and that are then used by `glocfn`, which is not expected to do any communication.

The user’s calling program should include the following elements:

- `#include "kinbbdpre.h"` for needed function prototypes and for type `KBBDData`
- `KBBDData p_data,`
- `machEnv = PVecInitMPI(comm, Nlocal, N, argc, argv),`
- `N_VMake(u, udata, machEnv),`
- `kmem = KINMalloc(N, F, . ),`
- `p_data = KBBDAlloc(Nlocal, mu, ml, . , glocfn, gcomm, . . ),` where the upper and lower half-bandwidths are `mu` and `ml`, respectively, and `glocfn` and `gcomm` are user-supplied functions
- `KINSpmr(kmem, maxi, maxlrst, msbpre, KBBDPrecon, KBBDPSol, userAtimes, p_data),` with the memory pointers `kmem` and `p_data` returned by the two previous calls, the parameters `(maxi, maxlrst, and msbpre)` and the names of the preconditioner routines (`KBBDPrecon, KBBDPSol`) supplied with the KINBBDDPRE module. If a user-supplied matrix-vector multiply routine, `userAtimes`, is supplied, it also is entered here
- `ier = KINSol(cvode_mem, u . . .),` to carry out the KINSOL solution
- `KBBDFree(p_data),` to free the KBBDPRE memory block
- `KINFree(kmem),` to free the KINSOL memory block
- `PVecFreeMPI(machEnv),` to free the KINSOL MPI memory block

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

- `KBBD_RPWSIZE(p_data) =` size of the real workspace (local to the current processor) used by KBBDPRE
- `KBBD_IPWSIZE(p_data) =` size of the integer workspace (local to the current processor) used by KBBDPRE
- `KBBD_NGE(p_data) =` cumulative number of `g` evaluations (calls to `glocfn`) so far

The costs associated with KBBDPRE also include `npe` LU factorizations, `npe` calls to `gcomm`, and `nps` banded backsolve calls, where `npe` and `nps` are optional KINSOL outputs.

Similar block-diagonal preconditioners could be considered with different treatment of the blocks \( P_m \). For example, incomplete LU factorization or an iterative method could be used instead of banded LU factorization.
6. The Fortran/C Interface Package. We anticipate that many users of KINSOL will work from existing Fortran application programs. To accommodate them, we have provided a set of interface routines that make the required connections to KINSOL with a minimum of changes to the application programs. Specifically, a Fortran/C interface package called FKINSOL is a collection of C language functions and header files which enables the user to write a main program and all user-supplied subroutines in Fortran and to use the C language KINSOL package. This package entails some compromises in portability, but we have kept these to a minimum by requiring fixed names for user-supplied routines, and by using a name-mapping scheme to set the names of externals in the Fortran/C linkages. The latter depends on two parameters set in a small header file.

Since a user cannot successfully link a program where any routine calls a Fortran routine not supplied, it is necessary that there be six choices for the FKINSPGMR routine. FKINSPGMR00 is found in fkinsol.c but the others are in separate files to simplify linking. Each calls the routine KINSpgmr (a C module) but with different options. The first of two suffix digits indicates whether the number of routines supplied is 0 (no preconditioning), 1 (preconditioner solve only), or 2 (both preconditioner setup and solve routines). The second digit indicates whether or not a userAtimes routine routine is supplied in Fortran. For example, if FKINSPGMR11 is called from the Fortran main, it will be necessary that the user supply as well the routines FPSOL and FATIMES. In this way, dummy routines named FPSOL, FATIMES, etc., are not required.

The Fortran/C interfaces have been tested on a Cray-T3D, a DEC ALPHA, and a cluster of Sun workstations.

A similar interface package, called FKINBBBD, has been written for the KINBBBDRE preconditioner module. It works in conjunction with the FKINSOL interface package. The additional user-callable functions here are KBBBDINITO and FKINBBBDINIT1, which interface with KBBBDAlloc and KINSpgmr, KINBBBDOPT, which accesses optional outputs, and KINBBBDFREE, which interfaces with KBBDFree. The two user-supplied Fortran subroutines required, in addition to KFUN to define F, are KLOCFN, which computes g(u), and KCOMMFN, which performs communications necessary for KLOCFN.

An overview of the Fortran interface and a skeleton program illustrating their use follow.

6.1. Overview of Fortran interface routines. The routines used to interface with a Fortran main program and with Fortran user-supplied routines are summarized below. Further details can be found in the header file fkinsol.h. Also, the user should check, and reset if necessary, the parameters in the file fcmixpar.h. The functions which are callable from the user's Fortran program are as follows:

- FKINITMPI interfaces with PVecInitMPI and is used to initialize the NVVECTOR module.
- FPKINMALLOC interfaces with KINMALLOC and is used to initialize KINSol.
- FKINSPGMR00, FKINSPGMR01, FKINSPGMR10, FKINSPGMR11, FKINSPGMR20, and FKINSPGMR21 interface with KINSpgmr when SPGMR has been chosen as the linear system solver (the only choice at present). These six interface routines correspond to the cases of no preconditioning, preconditioning with no saved matrix data, and preconditioning with saved matrix data, respectively, each without or with a user-
supplied Jacobian-vector multiply (FATIMES) routine. For example, FKINSPGMR11 uses conditioning but no setup routine (psolve but no precondset) and also the user has supplied a routine FATIMES that performs the Jacobian-vector multiply used in the GMRES solver.

- FKINSOL interfaces with KINSol
- FKINFREE interfaces with KINFree and is used to free memory allocated for CVode
- FKFREEMPI interfaces with PVecFreeMPI and is used to free memory allocated for MPI

Fortran interface modules and routines

<table>
<thead>
<tr>
<th>MODULE</th>
<th>Fortran-callable routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>FKINSOL</td>
<td>FKINITMPI, FKREEMPI, FPKINMALLOC,</td>
</tr>
<tr>
<td></td>
<td>FKINFREE, FKINSPGMR00, FINSOL</td>
</tr>
<tr>
<td>FKINSPGMR01</td>
<td>FKINSPGMR01</td>
</tr>
<tr>
<td>FKINSPGMR10</td>
<td>FKINSPGMR10</td>
</tr>
<tr>
<td>FKINSPGMR11</td>
<td>FKINSPGMR11</td>
</tr>
<tr>
<td>FKINSPGMR20</td>
<td>FKINSPGMR20</td>
</tr>
<tr>
<td>FKINSPGMR21</td>
<td>FKINSPGMR21</td>
</tr>
</tbody>
</table>

The user-supplied Fortran subroutines are as follows. The names of these routines are fixed and are case-sensitive.

- **KFUN** which defines the function, \( F \), that describes the system to be solved \( F(u) = 0 \)
- **KPSOL** which solves the preconditioner equation, and is required if preconditioning is used
- **KPRECO** which computes the preconditioner, and is required if preconditioning involves pre-computed matrix data
- **FATIMES** which performs a Jacobian-vector product paralleling the C routine user-Atimes
- **KLOCFN** which performs the local computation of the system function as required for the BBD preconditioner
- **KCOMMFN** which performs the communication of function values between processors as required for the BBD preconditioner

<table>
<thead>
<tr>
<th><strong>Routines to be provided by the user</strong> (*) indicates optional</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>KFUN</strong></td>
</tr>
<tr>
<td><strong>KPRECO</strong></td>
</tr>
<tr>
<td><strong>KPSOL</strong></td>
</tr>
<tr>
<td><strong>FATIMES</strong></td>
</tr>
<tr>
<td><strong>KLOCFN</strong></td>
</tr>
<tr>
<td><strong>KCOMMFN</strong></td>
</tr>
</tbody>
</table>
6.2. Skeleton of Fortran usage. The two summaries of usage in a Fortran context are brief but follow the pattern established above for the C interface.

Summary of Parallel Usage of KINSOL, using the Fortran interface
1. call MPI.INIT( ) Initialize MPI
2. call FKINITMPI(nlocal, neq, ier) Initialize the NVECTOR interface to MPI Here, nlocal and neq are the local and global sizes of vectors to be used
3. call MPI.COMM_SIZE(. ) or call MPI.COMM_RANK( . ) Optional calls to determine logical processor number and count, part of MPI, proper
4. call FPKINMALLOC(. ) Allocate space for KINSOL
5. call FKINSPGMR20( . ) Set up the linear solver The choice illustrated here is for both a setup and solve preconditioner routine to be supplied by the user in Fortran, but no user-supplied FATIMES routine
6. call FKINSOL( . ) Call KINSol, through the Fortran interface
7. call FKINFREE Free memory usage by KINSOL and its Fortran interface
8. call FKFREEMPI Free MPI interface

Summary of Serial Usage of KINSOL, using the Fortran interface
1. call FPKINMALLOC(. ) Allocate space for KINSOL
2. call FKINSPGMR20( . ) Set up the linear solver The choice illustrated here is for both a setup and solve preconditioner routine to be supplied by the user in Fortran, but no user-supplied FATIMES routine
3. call FKINSOL( . ) Call KINSol, through the Fortran interface
4. call FKINFREE Free memory usage by KINSOL and its Fortran interface

7. Example Problems. Although a trivial diagonal example is supplied with the distribution package, the following example, the so-called predator-prey PDE system, is more illustrative of the power of KINSOL with real problems This particular problem, outlined below, was solved by both a sequential and parallel implementation of KINSOL (kinxs.c and kinxp.c being the C program source) It was also solved using the Band-Block-Diagonal Preconditioner supplied with KINSOL (kinxbbd.c) The PDE problem to be solved is now briefly presented

This example problem is a model of a multi-species food web [1], in which mutual competition and/or predator-prey relationships in a spatial domain are simulated For this problem the dependent variable c replaces the generic dependent variable u used above Here we consider a model with s = 2p species, where both species 1, · · · , p (the prey) and p + 1, · · · , s (the predators) have infinitely fast reaction rates

\[
\begin{align*}
0 &= f_i(x, y, c) + d_i(c^i_{xx} + c^i_{yy}) \quad (i = 1, 2, \cdots, p), \\
0 &= f_i(x, y, c) + d_i(c^i_{xx} + c^i_{yy}) \quad (i = p + 1, \cdots, s),
\end{align*}
\]

with

\[
f_i(x, y, c) = c^i(b_i + \sum_{j=1}^{s} a_{ij} c^j)
\]

The interaction and diffusion coefficients (a_{ij}, b_i, d_i) could be functions of (x, y) in general The choices made for this test problem are for a simple model of p prey and p predator
species, arranged in that order in the vector $c$. We take the various coefficients to be as follows

\begin{equation}
\begin{aligned}
a_{ii} &= -1 \quad (\text{all } i) \\
a_{ij} &= -0.5 \times 10^{-6} \quad (i \leq p, j > p) \\
a_{ij} &= 10^{4} \quad (i > p, j \leq p)
\end{aligned}
\end{equation}

(all other $a_{ij} = 0$),

\begin{equation}
\begin{aligned}
b_{i} &= b_{i}(x, y) = (1 + \alpha xy) \quad (i \leq p) \\
b_{i} &= -(1 + \alpha xy) \quad (i > p)
\end{aligned}
\end{equation}

and

\begin{equation}
\begin{aligned}
d_{i} &= 1 \quad (i \leq p) \\
d_{i} &= 0.5 \quad (i > p)
\end{aligned}
\end{equation}

The domain is the unit square $0 \leq x, y \leq 1$. The boundary conditions are of Neumann type (zero normal derivatives) everywhere. The coefficients are such that a unique stable equilibrium is guaranteed to exist when $\alpha$ is zero [1]. Empirically, for (9) a stable equilibrium appears to exist when $\alpha$ is positive, although it may not be unique. In this problem we take $\alpha = 1$. The initial conditions used for this problem are taken to be constant functions by species type. These satisfy the boundary conditions and very nearly satisfy the constraints, given by

\begin{equation}
\begin{aligned}
c_{i} &= 116347 \quad (i = 1, \ldots, p) \\
c_{i} &= 349031 \quad (i = p + 1, \ldots, s)
\end{aligned}
\end{equation}

The PDE system (9) (plus boundary conditions) was discretized with central differencing on an $L \times L$ mesh, with the resulting nonlinear system has size $N = sL^2$.

The main program source solving this problem (kinxs_c) is given in its entirety in the Appendix. The output for this case is also included in the Appendix.

8. Availability. At present, the KINSOL package has not been released for general distribution. However, plans are in progress for a limited release, and interested potential users at DOE Laboratories can obtain KINSOL on request from Allan Taylor or Alan Hindmarsh (at agtaylor@llnl.gov or alanh@llnl.gov, resp).

REFERENCES


Appendix: Listing of Predator-Prey PDE Example Program.

```c
/*********************/
/* File: kinxp.c */
/* Programmers: Allan G Taylor and Alan C. Hindmarsh @ LLNL */
/* Version of 1 Dec 1997 */
/*********************/

Example problem for KINSol, parallel machine case
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

1 2 ns
cc = (c1, c2, ..., cn)

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.
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 local variable globalstrat.

The preconditioner matrix is a block-diagonal matrix based on the partial derivatives of the interaction terms f (in the above equation) only.

Execution: mpirun -np N -machinefile machines kinxp
(with N = NPEX*NPEY, total number of processors, see below)

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 (for a description of the time-dependent version of this test problem)

run command line: mpirun -np N -machinefile machines kinxp
where N = NPEX * NPEY is the number of processors to use.

*******************************************************************************

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "llnltyps.h" /* definitions of real, integer, boole, TRUE, FALSE*/
#include "kinsol.h" /* main KINSol header file */
#include "iterativ.h" /* contains the enum for types of preconditioning */
#include "kinspgrm.h" /* use KINSpgmr linear solver */
#include "dense.h" /* use generic DENSE solver for preconditioning */
#include "nvector.h" /* definitions of type N_Vector, macro N_VDATA */
#include "llnlmath.h" /* contains RSqrt and UnitRoundoff routines */
#include "mpi.h" /* MPI include file */

/* Problem Constants */

#define NUM_SPECIES 6 /* must equal 2*(number of prey or predators) number of prey = number of predators */
#define PI 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 /* MXSUB = number of x mesh points per subgrid */
#define MXSUB 10 /* MYSUB = number of y mesh points per subgrid */
#define MX (NPEX*MXSUB) /* number of grid points in the x-direction */
#define MY (NPEY*MYSUB) /* number of grid points in the y-direction */
#define NSMXSUB (NUM_SPECIES * (MXSUB+2))
#define NSMXSUBZ (NUM_SPECIES * MX + MY)
#define NEQ /* number of equations in the system */
#define AA RCONST(1.0) /* value of coefficient a, above eqns */
#define EE RCONST(10000) /* value of coefficient e, above eqns */
#define GG RCONST(0.5e-6) /* value of coefficient g, above eqns */
#define BB RCONST(1.0) /* value of coefficient b, above eqns */
#define DPREY RCONST(1.0) /* value of coefficient dprey, above eqns */
#define DPRED RCONST(0.5) /* value of coefficient dpred, above eqns */
#define ALPHA RCONST(1.0) /* value of coefficient alpha, above eqns */
#define AX RCONST(1.0) /* total range of x variable */
#define AY RCONST(1.0) /* total range of y variable */
#define FTOL RCONST(1e-7) /* ftol tolerance */
#define STOL RCONST(1e-13) /* stol tolerance */
#define THOUSAND RCONST(1000.0) /* one thousand */
#define ZERO RCONST(0.) /* 0 */
#define ONE RCONST(1.0) /* 1 */

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

/* IJ_Vptr is define in order to isolate the underlying 3-d structure of the
dependent variable vector from its underlying 1-d storage (an N_Vector).
IJ_Vptr returns a pointer to the location in vv corresponding to
ns = 0 , jx = i, jy = j */

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

/* Type : UserData
contains preconditioner blocks, pivot arrays, and problem constants */
typedef struct {
  real **P[MXSUB][MYSUB];
  integer *pivot[MXSUB][MYSUB];
  real **acoef, *bcoef;
}
N_Vector rates;
real *cox, *coy;
real cext[NUM_SPECIES * (MXSUB+2)*(MYSUB+2)];
integer my_pe, isubx, isuby, nsmsxsub, nsmsxsub2;

MPI_Comm comm;

real ax, ay, dx, dy;
real uround, sqruround;
integer Neq, mx, my, ns, np;
} *UserData;

/* Private Helper Functions */

static UserData AllocUserData(void);
static void InitUserData(integer my_pe, MPI_Comm comm, UserData data);
static void FreeUserData(UserData data);
static void SetInitialProfiles(N_Vector cc, N_Vector sc);
static void PrintOutput(integer my_pe, MPI_Comm comm, N_Vector cc);
static void PrintFinalStats(long int *iopt);
static void WebRate(real xx, real yy, real *cxy, real *ratesxy, void *f_data);
static real DotProduct(integer size, real *xl, real *x2);
static void BSend(MPI_Comm comm, integer my_pe, integer isubx, integer isuby,
integer dsizex, integer dsizey, real +cdata);
static void BRecvPost(MPI_Comm comm, MPI_Request request[], integer my_pe,
integer isubx, integer isuby,
integer dsizex, integer dsizey,
real *text, real *buffer);
static void BRecvWait(MPI_Request request[], integer isubx, integer isuby,
integer dsizex, real *text, real *buffer);
static void ccomm(integer Neq, real *cdata, UserData data);
static void fcalcpr(integer Neq, N_Vector cc, N_Vector fval,
void *f_data);

/* Functions Called by the KINSol Solver */

static void funcpr(integer Neq, N_Vector cc, N_Vector fval,
void *f_data);

static int Precondbd(integer Neq, N_Vector cc, N_Vector cscale,
N_Vector fval, N_Vector fscale,
N_Vector vtem,N_Vector vtemp1, SysFn func, real uround,
long int *nfePtr, void *P_data);
static int PSolvebd(integer Neq, N_Vector cc, N_Vector cscale, 
    N_Vector fval, N_Vector fscale, N_Vector vv, N_Vector ftem, 
    SysFn func, real uround, 
    long int *nfePtr, void *P-data);

 случая

 main(int argc, char *argv[])
{
    FILE *msgfile;
    integer Neq=NEQ;
    integer globalstrategy, i, local_N;
    real fnormtol, scstepTol, ropt[OPT_SIZE];
    long int iopt[OPT_SIZE];
    N_Vector cc, sc, constraints;
    UserData data;
    int iout, flag;
    int npelast = NPEX*NPEY-1;
    int my_pe, npes;
    bool optIn;
    void *mem;
    KINMem kmem;
    machEnvType machEnv;
    MPI_Comm comm;

    /* Allocate memory, and set problem data, initial values, tolerances */
    msgfile = fopen("PredPrey out","w");

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

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

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

/* allocate and initialize user data block */
data=(UserData)AllocUserData();
InitUserData(my_pe, comm, data);
machEnv = PVectInstMPI(comm, local_N, Neq, &argc, &argv);
if(machEnv==NULL) return(1);

/* example of changing defaults using iopt */
optIn = TRUE; for(i=0;i<KINSOL_IOPT_SIZE;i++)iopt[i]=0;
for(i=0;i<KINSOL_ROPT_SIZE;i++)ropt[i]=ZERO;
iopt[MXITER]=250;

/* choose global strategy */
globalstrategy = INEXACT_NEWTON;

/* allocate (initialize) vectors */
cc = N_VNew(Neq, machEnv);
sc = N_VNew(Neq, machEnv);
data->rates=N_VNew(Neq,machEnv);

constraints = N_VNew(Neq, machEnv);
N_VConSt(0.,constraints);

SetInitialProfiles(cc, sc);

fnormtol=FTOL; scsteptol=STOL;

/* Call KINMalloc to allocate KINSol memory block:
A pointer to KINSol problem memory is returned and stored in kmem.*/
mem = KINMalloc(Neq, msgfile, machEnv);
if(my_pe==0 & mem == NULL) { printf("KINMalloc failed."); return(1); }
kmem = (KINMem)mem;

/* Call KINSpgr to specify the KINSol linear solver KINSpgrm with solve
routines Precondbd and PSolvebd, and the pointer to
the user-defined block data */

KINSpgrm(kmem,
16, /* a zero in this position forces use of the KINSpgrm default
for maxl, dimension of the Krylov space*/
2, /* if zero in this position forces use of the KINSpgrm default
for maxlst, the max number of linear solver restarts allowed*/
0, /* a zero in this position forces use of the KINSpgrm default
for msbpre, the number of calls to the preconditioner allowed
without a call to the preconditioner setup routine */
    Precondbd, /* user-supplied preconditioner setup routine */
    PSolvebd, /* user-supplied preconditioner solve routine */
    NULL, /* user-supplied ATimes routine -- Null chosen here */
    data);

if(my_pe==0)printf("\n predator-prey test problem -- KINSol\n\n");

/* first, print out the problem size and then the
initial concentration profile */

if(my_pe==0){
    printf("Mesh dimensions %d X %d\n",MX,MY);
    printf("Total system size %d\n",Neq);
    printf("Preconditioning uses interaction-only block-diagonal matrix\n");
    printf("tolerance parameters: fnormtol = %g scsteptol = %g\n",
          fnormtol,scsteptol);

    printf("\nInitial profile of concentration\n");
}

if(my_pe==0 || my_pe==npelast) PrintOutput(my_pe, comm, cc);

/* call KINSol and print output concentration profile */

    flag = KINSol(kmem, /* KINSol memory block */
Neq, /* system size -- number of equations */
cc, /* solution cc of funcprpr(cc)=0 is desired */
funcprpr, /* function describing the system equations */
globalstrategy, /* global strategy choice */
sc, /* scaling vector, for the variable cc */
sval, /* scaling vector for function values fval */
fnormtol, /* tolerance on fnorm funcprpr(cc) for sol’n */
scsteptol, /* step size tolerance */
constraints, /* constraints vector */
optIn, /* optional inputs flat: TRUE or FALSE */
iopt, /* integer optional input array */
ropt, /* real optional input array */
data, /* pointer to user data */
msgfile, /* file pointer to message file */
machEnv); /* machEnv pointer */

if(my_pe==0){
    if (flag != KINSOL_SUCCESS) {
        printf("KINSol failed, flag=%d.\n", flag);
        return(flag); }

30
printf("\n\n\nComputed equilibrium species concentrations: \n\n");

if(my_pe==0 || my_pe==npelast)PrintOutput(my_pe, comm, cc);

/* cc values are available on each processor */
if(my_pe==0) PrintFinalStats(iopt);

/* Free memory and print final statistics */
N_VFree(cc);
N_VFree(sc);
N_VFree(constraints);
KINFree(kmem);
FreeUserDatea(data);

MPI_Finalize();
return(0);
}

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

/* Allocate memory for data structure of type UserData */
static UserData AllocUserData(void)
{
    int jx, jy;
    UserData data;

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

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

    return(data);
}
/* readability constants defined */

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

/**************************************************************************/

/* Load problem constants in data */

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

    data->mx = MX;
    data->my = MY;
    data->ns = NUM_SPECIES;
    data->np = NUM_SPECIES / 2;
    data->ax = AX;
    data->ay = AY;
    data->dx = (data->ax)/(MX-1);
    data->dy = (data->ay)/(MY-1);
    data->Neq= NEQ;
    data->my_pe = my_pe;
    data->comm = comm;
    data->isuby = my_pe / NPEX;
    data->isubx = my_pe - data->isuby*NPEX;
    data->nsnxsub = NUM_SPECIES * MXSUB;
    data->nsmxsubZ = NUM_SPECIES * (MXSUB+2);
    data->uround = UnitRoundoff();
    data->sqruround = RSqrt(data->uround);

    /* set up the coefficients a and b plus others found in the equations */
    np = data->np;
    dx2=(data->dx)*(data->dx); dy2=(data->dy)*(data->dy);

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

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

bcoef[i] = BB;
bcoef[i + np] = -BB;

cox[i] = DPREY/(dx2);
cox[i + np] = DPRED/(dx2);

coy[i] = DPREY/(dy2);
coy[i + np] = DPRED/(dy2);

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

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

denfree(acoef);
free(bcoef);
free(cox);
N_VFreeCdata-w&es);
free(data);
}
/* Set initial conditions in cc */

static void SetInitialProfiles(N_Vector cc, N_Vector sc)
{
    int i, jx, jy;
    real *ct1, *st1, *ct2, *st2;
    real ctemp[NUM_SPECIES], stemp[NUM_SPECIES];

    /* Initialize temporary arrays ctemp and stemp to be used
     * in the loading process */

    for(i=0;i<NUM_SPECIES;i++)
        if(i<NUM_SPECIES/2){
            ctemp[i]=RCONST(1.16347);
            stemp[i]=ONE;
        }
        else {
            ctemp[i]=RCONST(34903.1);
            stemp[i]=RCONST(0.00001);
        }

    /* Load initial profiles into cc and sc vector from temporary arrays */

    for (jy=0; jy < MYSUB; jy++) {
        for (jx=0; jx < MXSUB; jx++) {
            ct1 = IJ_Vptr(cc,jx,jy);
            ct2 = ctemp;
            st1 = IJ_Vptr(sc,jx,jy);
            st2 = stemp;
            for(i=0;i<NUM_SPECIES;i++){
                *ct1++=*ct2++;
                *st1++=*st2++;
            }
        }
    }
}

} /* end SetInitialProfiles */

/**************************************************************************/

/* Print sample of current cc values */

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

    /* Print sample of current cc values */

    for (i0=0; i0 < MYSUB; i0++) {
        for (jx=0; jx < MXSUB; jx++) {
            ct1 = IJ_Vptr(cc,jx,i0);
            ct2 = ct1;
            st1 = IJ_Vptr(sc,jx,i0);
            st2 = stemp;
            for(i=0;i<NUM_SPECIES;i++){
                *ct1++=*ct2++;
                *st1++=*st2++;
            }
        }
    }
}

} /* end PrintOutput */

/**************************************************************************/

*/
npelast = NPEX*NPEY - 1;

c = N_DATA(cc);

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

/* On PE 0, receive the cc values at top right, then print performance data
   and sampled solution values */
if(my_pe == 0) {
    if(npelast != 0)
        MPI_Recv(&tempc[0],NUM_SPECIES,PVEC_REAL_MPI_TYPE, npelast,0,comm,&status);
    printf("\n");
    printf("At bottom left:\n");
    for(is=0;is<NUM_SPECIES;is++){
        if((is%6)*6== is)printf("\n");
        printf(" %g",tempc[is]);
    }
    printf("\n\n");
    printf("At top right:\n");
    for(is=0;is<NUM_SPECIES;is++){
        if((is%6)*6 == is)printf("\n");
        printf(" %g",tempc[is]);
    }
    printf("\n\n");
}

/*****************************/

/* Print final statistics contained in iopt */

static void PrintFinalStats(long int *iopt)
{
    printf("\nFinal Statistics . \n\n");
    printf("nni = %5ld nli = %5ld\n", iopt[NNI], iopt[SPGMR_NLI]);
    printf("nfe = %5ld npe = %5ld\n", iopt[NFE], iopt[SPGMR_NPE]);
    printf("nps = %5ld ncfl = %5ld\n", iopt[SPGMR_NPS], iopt[SPGMR_NCFL]);
}
/* Routine to send boundary data to neighboring PEs */

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

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

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

    /* If isubx > 0, send data from left y-line of u (via bufleft) */
    if (isubx != 0) {
        for (ly = 0; ly < MYSUB; ly++) {
            offsetbuf = ly*NUM_SPECIES;
            offsetc = ly*dsizex;
            for (i = 0; i < NUM_SPECIES; i++)
                bufleft[offsetbuf+i] = cdata[offsetc+i];
        }
        MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
    }

    /* If isubx < NPEX-1, send data from right y-line of u (via bufright) */
    if (isubx != NPEX-1) {
        for (ly = 0; ly < MYSUB; ly++) {
            offsetbuf = ly*NUM_SPECIES;
            offsetc = offsetbuf*MXSUB + (MXSUB-1)*NUM_SPECIES;
            for (i = 0; i < NUM_SPECIES; i++)
                bufright[offsetbuf+i] = cdata[offsetc+i];
        }
        MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
    }
}
/* Routine to start receiving boundary data from neighboring PEs.
Notes:
1) buffer should be able to hold 2*NUM_SPECIES*MYSUB real entries, should be
passed to both the BRecvPost and BRecvWait functions, and should not
be manipulated between the two calls.
2) request should have 4 entries, and should be passed in both calls also. */

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

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

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

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

    /* If isubx < NPEX-1, receive data for right y-line of cext (via bufright) */
    if (isubx != NPEX-1) {
        MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
            my_pe+1, 0, comm, &request[3]);
    }
}
/* Routine to finish receiving boundary data from neighboring PEs. 
   Notes:
   1) buffer should be able to hold $2 \times \text{NUM\_SPECIES} \times \text{MYSUB}$ real entries, should be 
      passed to both the BRecvPost and BRecvWait functions, and should not 
      be manipulated between the two calls
   2) request should have 4 entries, and should be passed in both calls also. */

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

  dsizex2 = dsizex + 2*NUM_SPECIES;

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

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

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

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

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

    /* Copy the buffer to cext */
    for (ly = 0; ly < MYSUB; ly++) {

offsetbuf = 1y*NUM_SPECIES;
offsetcext = (1y+2)*dsizex2 - NUM_SPECIES;
for (i = 0; i < NUM_SPECIES; i++)
cext[offsetcext+i] = bufsight[offsetbuf+i];
}
}

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

static void ccomm(integer Neq, real *cdata, UserData data)
{
    real *cext, buffer[2*NUM_SPECIES*MYSUB];
    MPI_Comm comm;
    integer my_pe, isubx, isuby, nsmxsub, nsmysub;
    MPI_Request request[4];

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

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

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

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

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

static void fcalcprpr(integer Neq, N_Vector cc, N_Vector fval, void *f_data)
{
    real xx, yy, *cxy, *rxy, *fxy, dcydi, dcyui, dcxli, dcxri;
    real *cext, dely, delx, *cdata;
    integer i, j, is, ly;
    integer isubx, isuby, nsmxsub, nsmxsub2;
    integer shifty, offsetc, offsetce, offsetcl, offsetcr, offsetcd, offsetcu;
    UserData data;

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

    /* Get subgrid indices, data sizes, extended work array cext */

    isubx = data->isubx; isuby = data->isuby;
    nsmxsub = data->nsmxsub; nsmxsub2 = data->nsmxsub2;
    cext = data->cext;

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

    offsetc = 0;
    offsetce = nsmxsub2 + NUM_SPECIES;
    for (ly = 0; ly < MYSUB; ly++) {
        for (i = 0; i < nsmxsub; i++) cext[offsetce+i] = cdata[offsetc+i];
        offsetc = offsetc + nsmxsub;
        offsetce = offsetce + nsmxsub2;
    }

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

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

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

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

/* loop over all grid points, evaluating for each species at each */
delx = data->dx;
dely = data->dy;
shifty = (MXSUB+1)*NUM_SPECIES;
for(j=0; j<MYSUB; j++) {
    yy = dely*(j + isuby * MYSUB);
    for(i=0; i<MXSUB; i++) {
        xx = delx * ( i + isubx * MXSUB);
        cxy = IJ_Vptr(cc,i,j);
        rxy = IJ_Vptr(data->rates,i,j);
        fxy = IJ_Vptr(fval,i,j);
        WebRate(xx, yy, cxy, rxy, f_data);
        offsetc = (i+1)*NUM_SPECIES + (j+1)*NSMXSUB2;
        offsetcd = offsetc - shifty;
        offsetcu = offsetc + shifty;
        offsetcl = offsetc - NUM_SPECIES;
        offsetcr = offsetc + NUM_SPECIES;
        for(is=0; is<NUM_SPECIES; is++) {
            /* differencing in x */
            dcydi = cext[offsetc+is] - cext[offsetcd+is];
            dcyui = cext[offsetcu+is] - cext[offsetc+is];
            ...
/* differencing in y */

dcxli = cext[offsetc+isl] - cext[offsetcl+isl];
dcxri = cext[offsetcr+isl] - cext[offsetc+isl];

/* compute the value at xx, yy */

fxy[isl] = (coy)[isl] * (dcyui - dcydi) +
          (cox)[isl] * (dcxri - dcxli) + rxy[isl];

} /* end is loop */

} /* end of i or x loop */

} /* end of j or y loop */

} /* end of routine fcalcprpr */

/**************************** Functions Called by the KINSol Solver ***************************/

/* system function routine. Evaluate funcprpr(cc). First call ccomm to do
   communication of subgrid boundary data into cext. Then calculate funcprpr
   by a call to fcalcprpr */

static void funcprpr(integer Neq, N_Vector cc, N_Vector fval, void *f_data)
{
    real *cdata, *fvdata;
    UserData data;

    cdata = N_VDATA(cc);
    fvdata = N_VDATA(fval);
    data = (UserData) f_data;

    /* Call ccomm to do inter-processor communicaiton */
    ccomm (Neq, cdata, data);

    /* Call fcalc to calculate the system function */
    fcalcprpr (Neq, cc, fval, data);
}

51
/* Preconditioner setup routine. Generate and preprocess P. */

static int Precondbdc(integer Neq, N_Vector cc, N_Vector cscale,
    N_Vector fval, N_Vector fscale,
    N_Vector vtem, N_Vector vtemp1, SysFn func, real uround,
    long int *nfePtr, void *P-data)
{
    real r, r0, sqruround;
    real xx, yy, *cxy, *scxy, cctemp, **Pxy, *ratesxy, *Pxycol;
    real fac, perturb_rates[NUM_SPECIES];

    integer i, j, jx, jy, ret;

    UserData data;

data = (UserData)P-data;

    sqruround = data->sqruround;
    fac = N_VWL2Norm(fval, fscale);
    r0 = THOUSAND * uround * fac * Neq;

    if(r0 == ZERO) r0 = ONE;

    for(jy=0; jy<MYSUB; jy++)
    {
        yy = data->dy[(jy + data->isuby * MYSUB)];
        for(jx=0; jx<MXSUB; jx++)
        {
            xx = data->dx[(jx + data->isubx * MXSUB)];
            Pxy = (data->P)[jx][jy];
            cxy = IJ_Vptr(cc, jx, jy);
            scxy = IJ_Vptr(cscale, jx, jy);
            ratesxy = IJ_Vptr((data->rates), jx, jy);

            for(j=0; j<NUM_SPECIES; j++)
            {
                cctemp = cxy[j]; /* save the j,jx,jy element of cc */
                r = MAX(sqruround * ABS(cctemp), r0/scxy[j]);
                cxy[j] += r; /* perturb the j,jx,jy element of cc */
                fac = ONE/r;
            }
        }
    }
}

52
WebRate(xx, yy, cxy, perturb_rates, data);

Pxycol = Pxy[j];

for(i=0; i<NUM_SPECIES; i++) {
    Pxycol[i] = (perturb_rates[i] - ratesxy[i]) * fac;
}

/* restore j, jx, jy element of cc */
cxy[j] = cctemp;
}

/* lu decomposition of each block */
ret = gefa(Pxy, NUM_SPECIES, (data->pivot)[jx][jy]);

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

/* end of jy loop */

} /* end of routine Precondbd */

="/" Preconditioner solve routine /*

static int PSolvebd(integer Neq, N_Vector cc, N_Vector cscale, 
    N_Vector fval, N_Vector fscale, N_Vector vv, N_Vector ftem, 
    SysFn func, real wound, 
    long int *nfePtr, void *P_data)
{
    real **Pxy, *vxy;
    integer *pivot, jx, jy;
    UserData data;

    data = (UserData)P_data;

    for( jx=0; jx<MXSUB; jx++) {
        for(jy=0; jy<MYSUB; jy++) {
            /* for a given jx, jy block, do the inversion process */
/ vvxy is the address of the portion of the vector to which the inversion process is applied, and Pxy is the first address for the jx,jy block of P */
    pivot=(data->pivot)[jx][jy];
    Pxy = (data->P)[jx][jy];
    vxy = IJ_Vptr(vv,jx,jy);
    gesl(Pxy, NUM_SPECIES, pivot, vxy);

} /* end of jy loop */

} /* end of jx loop */

return(0);
}
/* end of PSolvebd */

/*************************************************************************/

static void WebRate(real xx, real yy, real *cxy, real *ratesxy, void *f_data)
{
    integer i;
    integer j;
    real fac;
    UserData data;

    data = (UserData)f_data;

    for(i=0;i<NUM_SPECIES;i++)
        ratesxy[i] = DotProd(NUM_SPECIES, cxy, acoef[i]);
    /* above, ratesxy is used as an intermediate array. see below */

    fac = ONE + ALPHA * xx * yy;

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

} /* end WebRate */

/*************************************************************************/

static real DotProd(integer size, real *xl, real *x2)
{
    integer i;
    real *xx1, *xx2, temp = ZEIJU;

54
xx1 = x1; xx2 = x2;
for(i=0; i<size; i++) temp += *xx1++ * *xx2++;
return(temp);
}
Sample output for the sample case KINXP

predator-prey test problem -- KINSol

Mesh dimensions 20 X 20
Total system size 2400
Preconditioning uses interaction-only block-diagonal matrix
tolerance parameters: fnormtol = 1e-07  scsteptol = 1e-13

Initial profile of concentration

At bottom left:

1.16347 1.16347 1.16347 34903 1 34903 1 34903.1

At top right:

1.16347 1 16347 1.16347 34903 1 34903.1 34903.1

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..

nmi = 68  nli = 1339
nfe = 1476  npe = 6
nps = 1407  ncf1 = 16