

Example Programs for KINSOL v7.8.0

SUNDIALS v7.8.0

Aaron M. Collier¹ and Radu Serban¹

¹*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory*

June 25, 2026



UCRL-SM-208116

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 Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed 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 Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

CONTRIBUTORS

The SUNDIALS library has been developed over many years by a number of contributors. The current SUNDIALS team consists of Cody J. Balos, David J. Gardner, Alan C. Hindmarsh, Daniel R. Reynolds, and Carol S. Woodward. We thank Radu Serban for significant and critical past contributions.

Other contributors to SUNDIALS include: Mustafa Aggul, James Almgren-Bell, Lawrence E. Banks, Peter N. Brown, George Byrne, Rujeko Chinomona, Scott D. Cohen, Aaron Collier, Keith E. Grant, Steven L. Lee, Shelby L. Lockhart, John Loffeld, Daniel McGreer, Yu Pan, Slaven Peles, Cosmin Petra, Steven B. Roberts, H. Hunter Schwartz, Jean M. Sexton, Dan Shumaker, Steve G. Smith, Shahbaj Sohal, Allan G. Taylor, Hilari C. Tiedeman, Chris White, Ting Yan, and Ulrike M. Yang.

Contents

1	Introduction	1
1.1	Serial examples	1
1.2	MPI examples	2
1.3	OpenMP example	2
1.4	MPI + CUDA example	2
1.5	C++ examples	3
1.6	Fortran examples	3
2	C Example Problems	5
2.1	A serial dense example: kinFerTron_dns	5
2.2	A serial Krylov example: kinFoodWeb_kry	8
2.3	A parallel example: kinFoodWeb_kry_bbd_p	10
3	C++ Example Problems	13
3.1	Parallel Matrix-Free Example: kin_heat2D_nonlin_p	13
3.2	Parallel Example Using hypre: kin_heat2D_nonlin_hypre_pfmng	14
4	Fortran Example Problems	17
4.1	A serial example: kinDiagon_kry_f2003	17
4.2	A parallel example: kin_diagon_kry_f2003	19
	Bibliography	21

Chapter 1

Introduction

KINSOL includes examples of many types, to illustrate the use of various nonlinear and linear solver options and vector implementations.

With the exception of “demo”-type example files, the names of the examples are generally of the form `[slv][PbName]_[strat]_[ls]_[prec]_[p]`, where:

- `[slv]` identifies the solver (in this case `kin`).
- `[PbName]` identifies the problem.
- `[strat]` identifies the strategy (absent if “none” or “linesearch”).
- `[ls]` identifies the linear solver module used.
- `[prec]` indicates the KINSOL preconditioner module used (only if applicable, for examples using a Krylov linear solver and the KINBBDPRE module, this will be `bbd`).
- `[p]` indicates an example using the MPI parallel vector.

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

Note

The examples in the KINSOL distribution are written in such a way as to compile and run for any combination of configuration options used during the installation of SUNDIALS. As a consequence, they contain portions of code that will not be typically present in a user program. For example, programs may make use of the variables `SUNDIALS_EXTENDED_PRECISION` and `SUNDIALS_DOUBLE_PRECISION` to test if the solver libraries were built in extended or double precision, and use the appropriate conversion specifiers in `printf` functions.

1.1 Serial examples

Supplied in the `examples/kinsol/serial` directory are the following serial examples (using the [serial vector](#)):

- `kinRoberts_fp` solves the backward Euler time step for a three-species chemical kinetics system, using the fixed point strategy.

- `kinFerTron_dns` solves the Ferraris-Tronconi problem.

This program solves the problem with the [dense linear solver](#) and uses different combinations of globalization and Jacobian update strategies with different initial guesses.

- `kinFerTron_klu` solves the same problem as in `kinFerTron_dns`, but uses the [KLU sparse direct linear solver](#).
- `kinRoboKin_dns` solves a nonlinear system from robot kinematics.

This program solves the problem with the [dense linear solver](#) and a user-supplied Jacobian routine.

- `kinRoboKin_slv` is the same as `kinRoboKin_dns` but uses the [SuperLU_MT sparse direct linear solver](#).
- `kinLaplace_bnd` solves a simple 2D elliptic PDE on a unit square.

This program solves the problem with the [banded direct linear solver](#).

- `kinLaplace_picard_bnd` is the same as `kinLaplace_bnd` but uses the Picard strategy.
- `kinLaplace_picard_kry` is the same as `kinLaplace_picard_bnd` but uses the [GMRES iterative linear solver](#).
- `kinFoodWeb_kry` solves a food web model.

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

- `kinKrylovDemo_ls` solves the same problem as `kinFoodWeb_kry`, but with three Krylov linear solvers: [GMRES](#), [BiCGSTAB](#), and [TFQMR](#).
- `kinAnalytic_fp.c` solves a small nonlinear system with known solution using the fixed-point iteration.

1.2 MPI examples

Supplied in the `examples/kinsol/parallel` directory are the following parallel examples (using the [MPI parallel vector](#)):

- `kinFoodWeb_kry_p` is a parallel implementation of `kinFoodWeb_kry`.
- `kinFoodWeb_kry_bbd_p` solves the same problem as `kinFoodWeb_kry_p`, with the [KINBBDPRE](#) band-block-diagonal preconditioner.

1.3 OpenMP example

Supplied in the `examples/kinsol/C_openmp` directory is an example `kinFoodweb_kry_omp`, which solves the same problem as `kinFoodweb_kry`, but using the [OpenMP vector](#).

1.4 MPI + CUDA example

Supplied in the `examples/kinsol/CUDA_mpi` directory is an example `kin_em_mpicuda`, which solves an expectation-maximization problem for mixture densities, using the [MPI+X](#) and [CUDA](#) vectors.

1.5 C++ examples

- The directory `examples/kinsol/CXX_parallel` contains two examples. `kin_heat2D_nonlin_p` solves a steady state 2D heat equation with an additional nonlinear term. This example is solved with fixed point iteration and illustrates the use of various orthogonalization methods with Anderson acceleration. `kin_em_p` solves the same problem as `kin_em_mpicuda` listed above.
- The directory `examples/kinsol/CXX_parhyp` contains two examples, `kin_heat2D_nonlin_hypre_pfm` and `kin_bratu2D_hyper_pfm`. These use the *hypre* PFMG preconditioner, fixed point iteration, and Anderson acceleration.

1.6 Fortran examples

The following are examples in Fortran, using the SUNDIALS Fortran interface modules, and are based on examples listed earlier:

- The directory `examples/kinsol/F2003_serial` contains four examples: `kinDiagon_kry_f2003` solves a simple diagonal test problem. `kinLaplace_bnd_f2003` solves the same problem as `kinLaplace_bnd`. `kinLaplace_picard_kry_f2003` solves the same problem as `kinLaplace_picard_kry`. `kinRoboKin_dns_f2003` solves the same problem as `kinRoboKin_dns`.
- The directory `examples/kinsol/F2003_parallel` contains one example, `kin_diagon_kry_f2003`, which solves the same problem as `kinDiagon_kry_f2003`, but using MPI.

Chapter 2

C Example Problems

2.1 A serial dense example: kinFerTron_dns

As an initial illustration of the use of the KINSOL package for the solution of nonlinear systems, we give a sample program called `kinFerTron_dns.c`. It uses the [dense linear solver](#) and the [serial vector](#) for the solution of the Ferraris-Tronconi test problem [1].

This problem involves a blend of trigonometric and exponential terms,

$$\begin{aligned} 0 &= 0.5 \sin(x_1 x_2) - 0.25 x_2 / \pi - 0.5 x_1, \\ 0 &= (1 - 0.25 / \pi)(e^{2x_1} - e) + e x_2 / \pi - 2e x_1, \end{aligned}$$

and is subject to the following constraints

$$\begin{aligned} x_{1 \min} &= 0.25 \leq x_1 \leq 1 = x_{1 \max} \\ x_{2 \min} &= 1.5 \leq x_2 \leq 2\pi = x_{2 \max} \end{aligned}$$

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

$$\begin{aligned} l_1 &= x_1 - x_{1 \min} \geq 0 \\ L_1 &= x_1 - x_{1 \max} \leq 0 \\ l_2 &= x_2 - x_{2 \min} \geq 0 \\ L_2 &= x_2 - x_{2 \max} \leq 0 \end{aligned}$$

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

Following the initial comment block, this program has a number of `#include` lines, which allow access to useful items in KINSOL header files:

- `kinsol.h` provides prototypes for the KINSOL functions to be called and also a number of constants that are to be used in setting input arguments and testing the return value of `KINSol()`.
- `nvector_serial.h` provides prototypes for the serial vector implementation.
- `sunmatrix_dense.h` provides the prototypes for the dense matrix implementation.
- `sunlinsol_dense.h` provides the prototypes for the dense linear solver implementation.

- `sundials_types.h` file provides the definition of `sunrealtype`. For now, it suffices to read `sunrealtype` as `double`.

Next, the program defines some problem-specific constants, which are isolated to this early location to make it easy to change them as needed.

The program prologue ends with prototypes of the user-supplied system function `func` and several helper functions.

The main program begins with creating the SUNDIALS context object (`sunctx`) which must be passed to all other SUNDIALS constructors. Then we allocated the user data structure, `data`, which contains two arrays with lower and upper bounds for x_1 and x_2 . Next, we create five serial vectors for the two different initial guesses (`u1` and `u2`), the solution vector (`u`), the scaling factors (`s`), and the constraint specifications (`c`).

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

The calls to `N_VNew_Serial()`, and also later calls to various `KIN***` functions, make use of a function, `check_flag`, which examines the return value and prints a message if there was a failure. The `check_flag` function was written to be used for any serial SUNDIALS application.

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

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

The solver is initialized with `KINInit()` call which specifies the system function `func` and provides the vector `u` which will be used internally as a template for cloning additional necessary vectors of the same type as `u`.

The call to `SUNDenseMatrix()` to creates the Jacobian matrix to use with the dense linear solver which is created by `SUNLinSol_Dense()`. Finally, both of these objects are attached to KINSOL by calling `KINSetLinearSolver()`.

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

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

The function `func` is a straightforward expression of the extended nonlinear system. It uses the `N_VGetArray-Pointer()` function to extract the data arrays of the vectors `u` and `f` and sets the components of `fdata` using the current values for the components of `udata`. See `KINSysFn` for a detailed specification of `func`.

The output generated by `kinFerTron_dns` is shown below.

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

(continues on next page)

(continued from previous page)

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

Exact Newton

Solution:

```
[x1,x2] = 0.299449  2.83693
```

Final Statistics:

```
nni =    3    nfe =    4
nje =    3    nfeD =   18
```

Exact Newton with line search

Solution:

```
[x1,x2] = 0.299449  2.83693
```

Final Statistics:

```
nni =    3    nfe =    4
nje =    3    nfeD =   18
```

Modified Newton

Solution:

```
[x1,x2] = 0.299449  2.83693
```

Final Statistics:

```
nni =   11    nfe =   12
nje =    2    nfeD =   12
```

Modified Newton with line search

Solution:

```
[x1,x2] = 0.299449  2.83693
```

Final Statistics:

```
nni =   11    nfe =   12
nje =    2    nfeD =   12
```

```
-----
Initial guess in middle of feasible region
```

```
[x1,x2] =      0.625    3.89159
```

Exact Newton

Solution:

```
[x1,x2] =      0.5    3.14159
```

Final Statistics:

```
nni =    5    nfe =    6
nje =    5    nfeD =   30
```

Exact Newton with line search

Solution:

```
[x1,x2] =      0.5    3.14159
```

Final Statistics:

```
nni =    5    nfe =    6
nje =    5    nfeD =   30
```

(continues on next page)

(continued from previous page)

```

Modified Newton
Solution:
  [x1,x2] =  0.5000003    3.1416
Final Statistics:
  nni =    12    nfe =    13
  nje =     2    nfeD =    12

Modified Newton with line search
Solution:
  [x1,x2] =  0.5000003    3.1416
Final Statistics:
  nni =    12    nfe =    13
  nje =     2    nfeD =    12

```

2.2 A serial Krylov example: kinFoodWeb_kry

We give here an example that illustrates the use of KINSOL with the GMRES Krylov method as the linear system solver.

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

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

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

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

The problem coefficients are given by

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

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

and

$$d_i = \begin{cases} 1 & i \leq n_p \\ 0.5 & i > n_p \end{cases}$$

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

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

Among the initial `#include` lines in this case is `sunlinsol_spgmr.h` which contains constants and function prototypes associated with the **GMRES linear solver**.

The main program calls `KINCreate()` and then calls `KINInit()` with the name of the user-supplied system function `func` and solution vector as arguments. The main program then calls a number of `KINSet*` routines to notify KINSOL of the user data pointer, the positivity constraints on the solution, and convergence tolerances on the system function and step size. It calls `SUNLinSol_SPGMR()` to create the linear solver, supplying the `maxl` value of 15 as the maximum Krylov subspace dimension. It then calls `KINSetLinearSolver()` to attach this solver module to KINSOL. Next, a maximum value of `maxlrst = 2` restarts is imposed through a call to `SUNLinSol_SPGMRSetMaxRestarts()`. Finally, the user-supplied preconditioner setup and solve functions, `PrecSetupBD` and `PrecSolveBD`, are specified through a call to `KINSetPreconditioner()`. The data pointer passed to `KINSetUserData()` is passed to `PrecSetupBD` and `PrecSolveBD` whenever these are called.

Next, `KINSol()` is called to solve the system, the return value is tested for error conditions, and the approximate solution vector is printed via a call to `PrintOutput`. After that, `PrintFinalStats` is called to get and print final statistics, and memory is freed by calls to `N_VDestroy()`, `FreeUserData`, `KINFree()`, and `SUNContext_Free()`. The statistics printed are the total numbers of nonlinear iterations (`nni`), `func` evaluations (`nfe`, excluding those for *Jv* product evaluations), `func` evaluations for *Jv* evaluations (`nfeSG`), linear (Krylov) iterations (`nli`), preconditioner evaluations (`npe`), and preconditioner solves (`nps`). See the **Optional output functions** section for more information.

Mathematically, the dependent variable has three dimensions: species number, *x* mesh point, and *y* mesh point. The macro `IJ_Vptr` isolates the translation from three dimensions to the one-dimensional contiguous array of components under the serial vector. This macro allows for clearer code and makes it easy to change the underlying layout of the three-dimensional data.

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

The program `kinFoodWeb_kry.c` uses the “small” dense functions for all operations on the 6×6 preconditioner blocks. Thus it includes `sundials_dense.h`, and calls the small dense matrix functions `SUNDlsMat_newDenseMat`, `SUNDlsMat_denseGETRF`, and `SUNDlsMat_denseGETRS`. The small dense functions are generally available for KINSOL user programs (for more information, see the comments in the header file `sundials_dense.h`).

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

The output generated by `kinFoodWeb_kry` is shown below. Note that the solution involved 9 Newton iterations, with an average of about 37 Krylov iterations per Newton iteration.

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

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

```
Flag globalstrategy = 0 (0 = None, 1 = Linesearch)
Linear solver is SPGMR with maxl = 15, maxlrst = 2
Preconditioning uses interaction-only block-diagonal matrix
Positivity constraints imposed on all components
Tolerance parameters: fnormtol = 1e-07  scsteptol = 1e-13
```

```
Initial profile of concentration
```

(continues on next page)

(continued from previous page)

```
At all mesh points:  1 1 1   30000 30000 30000
```

```
Computed equilibrium species concentrations:
```

```
At bottom left:
```

```
1.16428 1.16428 1.16428 34927.5 34927.5 34927.5
```

```
At top right:
```

```
1.25797 1.25797 1.25797 37736.7 37736.7 37736.7
```

```
Final Statistics..
```

```
nni   =    9   nli   =   329
nfe   =   10   nfeSG =   338
nps   =   338   npe   =    1   ncfl  =    6
```

2.3 A parallel example: kinFoodWeb_kry_bbd_p

In this example, `kinFoodWeb_kry_bbd_p`, we solve the same problem as with `kinFoodWeb_kry` above, but in parallel, and instead of supplying the preconditioner we use the `KINBBDPRE` preconditioner.

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

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

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

The function `func_local` is also passed as the `gloc` argument to `KINBBDPrecInit()`. Since all communication needed for the evaluation of the local approximation of f used in building the band-block-diagonal preconditioner is already done for the evaluation of f in `func`, a NULL pointer is passed as the `gcomm` argument to `KINBBDPrecInit()`.

The main program resembles closely that of the `kinFoodWeb_kry` example, with particularization arising from the use of the `MPI parallel vector`. It begins by initializing MPI and obtaining the total number of processes and the rank of the local process. The local length of the solution vector is then computed as $\text{NUM_SPECIES} * \text{MXSUB} * \text{MYSUB}$. Distributed vectors are created by calling the constructor `N_VNew_Parallel()` with the MPI communicator and the local and global problem sizes as arguments. All output is performed only from the process with ID equal to 0. Finally, after `KINSol()` is called and the results are printed, all memory is deallocated, and the MPI environment is terminated by calling `MPI_Finalize`.

The output generated by `kinFoodWeb_kry_bbd_p` is shown below. Note that 9 Newton iterations were required, with an average of about 52 Krylov iterations per Newton iteration.

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

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

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

Flag globalstrategy = 0 (0 = None, 1 = Linesearch)
Linear solver is SPGMR with maxl = 20, maxlrst = 2
Preconditioning uses band-block-diagonal matrix from KINBBDPRE
  Difference quotient half-bandwidths: mudq = 11, mldq = 11
  Retained band block half-bandwidths: mukeep = 6, mlkeep = 6
Tolerance parameters: fnormtol = 1e-07  scsteptol = 1e-13

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

Computed equilibrium species concentrations:

At bottom left:
  1.165 1.165 1.165 34949 34949 34949

At top right:
  1.25552 1.25552 1.25552 37663.2 37663.2 37663.2

Final Statistics..
nni   =    9   nli   =   464
nfe   =   10   nfeSG =   473
nps   =   473   npe   =    1   ncfl  =    6
```


Chapter 3

C++ Example Problems

3.1 Parallel Matrix-Free Example: kin_heat2D_nonlin_p

As an illustration using KINSOL for the solution of nonlinear systems in parallel, we give a sample program called `kin_heat2D_nonlin_p.cpp`. It uses the KINSOL fixed-point (KIN_FP) iteration with Anderson Acceleration and the [MPI parallel vector](#) for the solution of a steady-state 2D heat equation with an additional nonlinear term defined by $c(u)$:

$$b = \nabla \cdot (D \nabla u) + c(u) \quad \text{in } \mathcal{D} = [0, 1] \times [0, 1]$$

where D is a diagonal matrix with entries k_x and k_y for the diffusivity in the x and y directions, respectively. The boundary conditions are

$$u(0, y) = u(1, y) = u(x, 0) = u(x, 1) = 0.$$

We chose the analytical solution to be

$$u_{\text{exact}} = u(x, y) = \sin^2(\pi x) \sin^2(\pi y)$$

Hence, we define the static term b as follows

$$b = k_x 2\pi^2 (\cos^2(\pi x) - \sin^2(\pi x)) \sin^2(\pi y) + k_y 2\pi^2 (\cos^2(\pi y) - \sin^2(\pi y)) \sin^2(\pi x) + c(u_{\text{exact}})$$

The spatial derivatives are computed using second-order centered differences, with the data distributed over $n_x \times n_y$ points on a uniform spatial grid. The problem is set up to use spatial grid parameters $n_x = 64$ and $n_y = 64$, with heat conductivity parameters $k_x = 1.0$ and $k_y = 1.0$.

This problem is solved via a fixed point iteration with Anderson acceleration, where the fixed point is function formed by adding u to both sides of the equation, i.e.,

$$b + u = \nabla \cdot (D \nabla u) + c(u) + u,$$

so that the fixed point function is

$$G(u) = \nabla \cdot (D \nabla u) + c(u) + u - b.$$

The problem is run using a tolerance of 10^{-8} , and a starting vector containing all ones. This example highlights the use of the various orthogonalization routine options within Anderson Acceleration, passed to the example problem via the `--orthaa` flag. Available options include 0 (KIN_ORTH_MGS), 1 (KIN_ORTH_ICWY), 2 (KIN_ORTH_CGS2), and 3 (KIN_ORTH_DCGS2).

The following tables contain all available input parameters when running the example problem.

3.1.1 Optional Input Parameter Flags

Flag	Description
--mesh <nx> <ny>	mesh points in the x and y directions
--np <npx> <npy>	number of MPI processes in the x and y directions
--domain <xu> <yu>	domain upper bound in the x and y direction
--k <kx> <ky>	diffusion coefficients
--rtol <rtol>	relative tolerance
--maa <maa>	number of previous residuals for Anderson Acceleration
--damping <damping>	damping parameter for Anderson Acceleration
--orthaa <orthaa>	orthogonalization routine used in Anderson Acceleration
--maxits <maxits>	max number of iterations
--c <cu>	nonlinear function choice (integer between 1 - 17)
--timing	print timing data
--help	print available input parameters and exit

3.1.2 Input Parameter Flags for Nonlinear Function $c(u)$

Flag	Function
--c 1	$c(u) = u$
--c 2	$c(u) = u^3 - u$
--c 3	$c(u) = u - u^2$
--c 4	$c(u) = e^u$
--c 5	$c(u) = u^4$
--c 6	$c(u) = \cos^2(u) - \sin^2(u)$
--c 7	$c(u) = \cos^2(u) - \sin^2(u) - e^u$
--c 8	$c(u) = e^u u^4 - u e^{\cos(u)}$
--c 9	$c(u) = e^{(\cos^2(u))}$
--c 10	$c(u) = 10(u - u^2)$
--c 11	$c(u) = -13 + u + ((5 - u)u - 2)u$
--c 12	$c(u) = \sqrt{5}(u - u^2)$
--c 13	$c(u) = (u - e^u)^2 + (u + u \sin(u) - \cos(u))^2$
--c 14	$c(u) = u + u e^u + u e^{-u}$
--c 15	$c(u) = u + u e^u + u e^{-u} + (u - e^u)^2$
--c 16	$c(u) = u + u e^u + u e^{-u} + (u - e^u)^2 + (u + u \sin(u) - \cos(u))^2$
--c 17	$c(u) = u + u e^{-u} + e^u(u + \sin(u) - \cos(u))^3$

3.2 Parallel Example Using hypre: kin_heat2D_nonlin_hypre_pfmng

As an illustration of the use of the KINSOL package for the solution of nonlinear systems in parallel and using *hypre* linear solvers, we give a sample program called `kin_heat2D_nonlin_hypre_pfmng.cpp`. It uses the KINSOL fixed-point (KIN_FP) iteration with Anderson Acceleration and the [MPI parallel vector](#) for the solution of a steady-state 2D heat equation with an additional nonlinear term defined by $c(u)$:

$$b = \nabla \cdot (D \nabla u) + c(u) \quad \text{in } \mathcal{D} = [0, 1] \times [0, 1]$$

where D is a diagonal matrix with entries k_x and k_y for the diffusivity in the x and y directions, respectively. The boundary conditions are

$$u(0, y) = u(1, y) = u(x, 0) = u(x, 1) = 0.$$

We chose the analytical solution to be

$$u_{\text{exact}} = u(x, y) = \sin^2(\pi x) \sin^2(\pi y)$$

Hence, we define the static term b as follows

$$b = k_x 2\pi^2 (\cos^2(\pi x) - \sin^2(\pi x)) \sin^2(\pi y) + k_y 2\pi^2 (\cos^2(\pi y) - \sin^2(\pi y)) \sin^2(\pi x) + c(u_{\text{exact}})$$

The spatial derivatives are computed using second-order centered differences, with the data distributed over $n_x \times n_y$ points on a uniform spatial grid. The problem is set up to use spatial grid parameters $n_x = 64$ and $n_y = 64$, with heat conductivity parameters $k_x = 1.0$ and $k_y = 1.0$.

This problem is solved via a fixed point iteration with Anderson acceleration, where the fixed point function is formed by implementing the Laplacian as a matrix-vector product,

$$b = Au + c(u)$$

and solving for u results in the fixed point function

$$G(u) = A^{-1}(b - c(u)).$$

The problem is run using a tolerance of 10^{-8} , and a starting vector containing all ones. The linear system solve is executed using the [PCG linear solver](#) with the *hypre* PFMG preconditioner. The setup of the linear solver can be found in the `Setup_LS` function, and setup of the *hypre* preconditioner can be found in the `Setup_Hypre` function. This example highlights the use of the various orthogonalization routine options within Anderson Acceleration, passed to the example problem via the `--orthaa` flag. Available options include 0 (KIN_ORTH_MGS), 1 (KIN_ORTH_ICWY), 2 (KIN_ORTH_CGS2), and 3 (KIN_ORTH_DCGS2).

All input parameter flags available for the previous example are also available for this problem. In addition, all runtime flags controlling the linear solver and *hypre* related parameters are set using the flags in the following table.

3.2.1 Optional Input Parameter Flags for hypre

Flag	Description
<code>--lsinfo</code>	output residual history for PCG
<code>--linitors <linitors></code>	max number of iterations for PCG
<code>--epslin <epslin></code>	linear tolerance for PCG
<code>--pfmg_relax <pfmg_relax></code>	relaxation type in PFMG
<code>--pfmg_nrelax <pfmg_nrelax></code>	pre/post relaxation sweeps in PFMG

Chapter 4

Fortran Example Problems

The Fortran example problem programs supplied with the KINSOL package are all written using the Fortran 2003 standard, use double precision arithmetic, and 32- or 64-bit integers for indexing. See the [Fortran](#) section in the KINSOL User Documentation for details.

4.1 A serial example: kinDiagon_kry_f2003

The kinDiagon_kry_f2003 program solves the problem

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

using the [serial vector](#) and the [GMRES linear solver](#) with a diagonal preconditioner.

The kinDiagonKry_mod module includes the problem parameters (e.g., `neq = N`) used in subroutines and functions contained in the module. The subroutines and function defined in the module are:

- The subroutine `init` sets the components of the initial guess vector (`sunvec_u`) to the values $u_i = 2i$, all the components of the scaling vector (`sunvec_s`) are set to `1.0` (i.e., no scaling is done), and the components of the constraints vector (`sunvec_c`) are set to `0.0` (i.e., no inequality constraints are imposed on the solution vector).
- The function `func` defines the user-supplied nonlinear residual $F(u)$.
- The functions `kpsetup` and `kpsolve` are the preconditioner setup and solve functions, respectively. `kpsetup` fills the array `p` with an approximation to the reciprocals of the Jacobian diagonal elements which are then used in `kpsolve` to solve the preconditioner linear system $Px = v$ by simple multiplication.

The main program starts with a number of `use` lines, which allow access to KINSOL (`fkinsol_mod`), the serial vector (`fnvector_serial_mod`), and the GMRES linear solver (`fsunlin_sol_spgmr_mod`). This is followed by a number of problem parameters for configuring the linear and nonlinear solver. In particular, the maximum number of iterations between calls to the preconditioner setup routine (`msbpre = 5`), the tolerance for stopping based on the function norm (`fnormtol = 1.0d-5`), and the tolerance for stopping based on the step length (`scsteptol = 1.0d-4`) are specified. After the problem parameters are declarations for local variables including SUNDIALS vector, matrix, and linear solver objects.

After printing the problem description, the main program creates the simulation context (`sunctx`) that is passed to all SUNDIALS constructors. The program then creates serial solution, scaling, and constraint vectors using `FN_VNew_Serial` and sets the vector values by calling the `init` subroutine.

The KINSOL solver is created by calling `FKINCreate` and initialized with `FKINInit` which takes as input the `c_funloc` of the Fortran residual function (`func`), a vector to use as a template to create internal workspace (`sunvec_u`), and the SUNDIALS context. Then solver options are specified by calling various `FKINSet` functions. In particular,

the maximum number of iterations between calls to the preconditioner setup routine (FKINSetMaxSetupCalls), the tolerance for stopping based on the function norm (FKINSetFuncNormTol), and the tolerance for stopping based on the step length (FKINSetScaledStepTol).

Next, the GMRES linear solver is created with FSUNLinSol_SPGMR specifying the maximum Krylov subspace dimension (maxl = 10) and right preconditioning (prectype = SUN_PREC_RIGHT). The linear solver is then attached to KINSOL with FKINSetLinearSolver. The input matrix (sunmat_J) is assigned to null for the matrix-free linear solver. The maximum number of restarts allowed for GMRES is then updated to maxlrst = 2 by calling FSUNLinSol_SPGMRSetMaxRestarts. The preconditioner functions are then attached by calling FKINSetPreconditioner and passing the c_funloc of kpsetup and kpsolve.

The solution of the nonlinear system is obtained after a successful return from FKINSol, which is then printed using the PrintOutput subroutine. Solver statistics are then output using the PrintFinalStats function which calls various FKINGet functions. Finally, the memory allocated for the KINSOL, the linear solver, vectors, and context are released by calling FKINFree, FSUNLinSolFree, FN_VDestroy, and FSUNContext_Free, respectively.

The following is sample output from kinDiagon_kry_f2003.

Example program kinDiagon_kry_f2003:

This example demonstrates using the KINSOL Fortran interface to solve a diagonal algebraic system using a Newton-Krylov method with a diagonal preconditioner in a serial environment.

Problem size: 128

```

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

```

(continues on next page)

(continued from previous page)

```

117 117.0000000 118.0000000 119.0000000 120.0000000
121 121.0000000 122.0000000 123.0000000 124.0000000
125 125.0000000 126.0000000 127.0000000 128.0000000

```

Final Statistics..

```

nni      =      7      nli      =     21
nfe      =      8      npe      =      2
nps      =     28      nlcf     =      0

```

4.2 A parallel example: kin_diagon_kry_f2003

The program `kin_diagon_kry_f2003` is a straightforward modification of `kinDiagon_kry_f2003` to use the MPI vector.

After initialing MPI, the MPI parallel vectors are created using `FN_VNew_Parallel` with the default MPI communicator and the local and global vector sizes as its first three arguments. The rank of the local process, `myid`, is used in both the initial guess and the system function, inasmuch as the global and local indices to the vector `u` are related by `iglobal = ilocal + mype*nlocal`. In other respects, the problem setup (KINSOL initialization, linear solver creation and specification, etc.) and solution steps are the same as in `kinDiagon_kry_f2003`. Upon successful return from `FKINSOL`, the solution on all ranks is printed and statistics are output from the root processes. Finally, the allocated memory is released and the MPI environment is finalized.

For this simple example, no inter-process communication is required to evaluate the nonlinear system function or the preconditioner. As a consequence, the user-supplied routines `func`, `kpsetup`, and `kpsolve` are basically identical to those in `kinDiagon_kry_f2003`.

The following is sample output from `kin_diagon_kry_2003` using 4 MPI ranks.

Example program `kinDiagon_kry_f2003`:

This example demonstrates using the KINSOL Fortran interface to solve a diagonal algebraic system using a Newton-Krylov method with a diagonal preconditioner in a MPI parallel environment.

Problem size: 128

Number of procs: 4

```

 1  1.0000000  2.0000000  3.0000000  4.0000000
 5  5.0000000  6.0000000  7.0000000  8.0000000
 9  9.0000000 10.0000000 11.0000000 12.0000000
13 13.0000000 14.0000000 15.0000000 16.0000000
17 17.0000000 18.0000000 19.0000000 20.0000000
21 21.0000000 22.0000000 23.0000000 24.0000000
25 25.0000000 26.0000000 27.0000000 28.0000000
29 29.0000000 30.0000000 31.0000000 32.0000000
33 33.0000000 34.0000000 35.0000000 36.0000000
37 37.0000000 38.0000000 39.0000000 40.0000000
41 41.0000000 42.0000000 43.0000000 44.0000000
45 45.0000000 46.0000000 47.0000000 48.0000000
49 49.0000000 50.0000000 51.0000000 52.0000000
53 53.0000000 54.0000000 55.0000000 56.0000000

```

(continues on next page)

(continued from previous page)

```

57 57.000000 58.000000 59.000000 60.000000
61 61.000000 62.000000 63.000000 64.000000
65 65.000000 66.000000 67.000000 68.000000
69 69.000000 70.000000 71.000000 72.000000
73 73.000000 74.000000 75.000000 76.000000
77 77.000000 78.000000 79.000000 80.000000
81 81.000000 82.000000 83.000000 84.000000
85 85.000000 86.000000 87.000000 88.000000
89 89.000000 90.000000 91.000000 92.000000
93 93.000000 94.000000 95.000000 96.000000
97 97.000000 98.000000 99.000000 100.000000
101 101.000000 102.000000 103.000000 104.000000
105 105.000000 106.000000 107.000000 108.000000
109 109.000000 110.000000 111.000000 112.000000
113 113.000000 114.000000 115.000000 116.000000
117 117.000000 118.000000 119.000000 120.000000
121 121.000000 122.000000 123.000000 124.000000
125 125.000000 126.000000 127.000000 128.000000

Final Statistics..

nni      =      7      nli      =     21
nfe      =      8      npe      =      2
nps      =     28      nlcf     =      0

```

Bibliography

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