Example Programs for KINSOL v6.4.1

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

October 31, 2022

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.

Approved for public release; further dissemination unlimited

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: 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, Slaven Peles, Cosmin Petra, H. Hunter Schwartz, Jean M. Sexton, Dan Shumaker, Steve G. Smith, Allan G. Taylor, Hilari C. Tiedeman, Chris White, Ting Yan, and Ulrike M. Yang.

Contents

1 Introduction

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

The KINSOL distribution contains examples of types: serial C examples, parallel C examples, serial and parallel FORTRAN examples, and an OpenMP example. With the exception of "demo"-type example files, the names of all the examples distributed with SUNDIALS are of the form `[slv][PbName]_[strat]_[ls]_[prec]_[p]`, where

[**slv**] identifies the solver (for KINSOL examples this is `kin`, while for FKINSOL examples, this is `fkin`);

[**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 parallel vector module NVECTOR_PARALLEL.

The following lists summarize all examples distributed with KINSOL.

Supplied in the *srcdir*/**examples/kinsol/serial** directory are the following serial examples (using the NVECTOR_SERIAL module):

- `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 SUNLINSOL_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 sparse direct solver KLU via the SUNLINSOL_KLU module.

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

- `kinRoboKin_slu` is the same as `kinRoboKin_dns` but uses the SuperLUMT sparse direct linear solver via the SUNLINSOL_SUPERLUMT module.

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

- `kinLaplace_picard_bnd` is the same as `kinLaplace_bnd` but uses the Picard strategy.

- `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 SUNLINSOL_SPGMR linear solver module 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: SUNLINSOL_SPGMR, SUNLINSOL_SPBCGS, and SUNLINSOL_SPTFQMR.

Supplied in the *srcdir*/`examples/kinsol/parallel` directory are the following parallel examples (using the NVECTOR_PARALLEL module):

- `kinFoodWeb_kry_p` is a parallel implementation of `kinFoodWeb_kry`.

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

As part of the FKINSOL module, in the directories *srcdir*/`examples/kinsol/fcmix_serial` and *srcdir*/`examples/kinsol/fcmix_parallel`, respectively, are the following examples for the FORTRAN-C interface:

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

- `fkinDiagon_kry_p` is a parallel implementation of `fkinDiagon_kry`.

Supplied in directory *srcdir*/`examples/kinsol/C_openmp` is an example using the OpenMP NVECTOR module. `kin_FoodWeb_kry_omp` solves the same problem as `kin_FoodWeb_kry` but uses the OpenMP module.

Supplied in directory *srcdir*/`examples/kinsol/CXX_parallel` is an example using the NVECTOR_PARALLEL module. `kin_heat2D_nonlin_p` solves a steady-state 2D heat equation with an additional nonlinear term. This problem is solved via fixed point iteration with Anderson acceleration. This example highlights the availability of various orthogonalization methods for use within Anderson acceleration.

Supplied in directory *srcdir*/`examples/kinsol/CXX_parhyp` are examples using the NVECTOR_PARALLEL module and incorporating use of *hypre* preconditioners and solvers. These example problems are solved via fixed point iteration with Anderson acceleration. They highlight the availability of various orthogonalization methods for use within Anderson acceleration.

- `kin_heat2D_nonlin_hypre_pfmg` solves the same problem as `kin_heat2D_nonlin_p` but with a different fixed point iteration setup requiring a linear solve that uses *hypre*'s PFMG preconditioner.

- `kin_bratu2D_hypre_pfmg` solves a 2D Bratu equation [3] requiring a linear solve that uses *hypre*'s PFMG preconditioner.

In the following 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%.

In the descriptions below, we make frequent references to the KINSOL User Document [1]. All citations to specific sections (e.g. §**??**) are references to parts of that User Document, unless explicitly stated otherwise.

Note. The examples in the KINSOL distribution are written in such a way as to compile and run for any combination of configuration options used during the installation of SUNDIALS (see Appendix **??** in the User Guide). As a consequence, they contain portions of code that will not be typically present in a user program. For example, all C example programs make use of the 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.

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 KINSOL dense linear solver module SUNLINSOL_DENSE and the NVECTOR_SERIAL module (which provides a serial implementation of NVECTOR) for the solution of the Ferraris-Tronconi test problem [2].

This problem involves a blend of trigonometric and exponential terms:

$$0 = 0.5\sin(x_1 x_2) - 0.25 x_2/\pi - 0.5 x_1$$
$$0 = (1 - 0.25/\pi)(e^{2x_1} - e) + ex_2/\pi - 2ex_1$$

subject to

$$x_{1\min} = 0.25 \leq x_1 \leq 1 = x_{1\max}$$
$$x_{2\min} = 1.5 \leq x_2 \leq 2\pi = x_{2\max}.$$

(1)

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:

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

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 CVODE header files. The `kinsol.h` file provides prototypes for the KINSOL functions to be called (excluding the linear solver selection function), and also a number of constants that are to be used in setting input arguments and testing the return value of `KINSol`. The `nvector_serial.h` file is the header file for the serial implementation of the NVECTOR module and includes definitions of the `N_Vector` type, a macro to access vector components, and prototypes for the serial implementation specific machine environment memory allocation and freeing functions. The `sunmatrix_dense.h` file provides the prototype for the `SUNDenseMatrix` function. The `sunlinsol_dense.h` file provides the prototype for the `SUNLinSol_Dense` function. The `sundials_types.h` file provides the definition of the type `realtype` (see §**??** for details). For now, it suffices to read `realtype` as `double`. Finally, `sundials_math.h` is included for the definition of the exponential function `RExp`.

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

The `main` program begins with some dimensions and type declarations, including use of the type `N_Vector`, initializations, and allocation and definitions for the user data structure `data` which contains two arrays with lower and upper bounds for x_1 and x_2. Next, we create five serial vectors of type `N_Vector` for the two different initial guesses, the solution vector `u`, the scaling factors, and the constraint specifications.

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

The calls to `N_VNew_Serial`, and also later calls to various `KIN***` functions, make use of a private 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.

Solver memory is allocated through the call to `KINInit` 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 use of the dense linear solver is specified by calling `SUNDenseMatrix` to create the template Jacobian matrix (which also specifies the problem size `NEQ`), then calling `SUNLinSol_Dense` to create the dense-direct linear solver, and finally calling `KINSetLinearSolver` to attach these to KINSOL.

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_Serial` from the NVECTOR_SERIAL module), with and without globalization through line search (specified by setting `glstr` to `KIN_LINESEARCH` and `KIN_NONE`, respectively), and applying either an exact or a modified Newton method. The switch from exact to modified Newton is done by changing the number of nonlinear iterations after which a Jacobian evaluation is enforced, a value `mset= 1` thus resulting in re-evaluating the Jacobian at every single 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 private function `SolveIt` which calls the main solver function `KINSol` after first setting the optional input `mset`. After a successful return from `KINSol`, the solution $[x_1, x_2]$ and some solver statistics are printed.

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

The output generated by `kinFerTron_dns` is shown below.

---------------------- `kinFerTron_dns` sample output ----------------------

```
 Ferraris and Tronconi test problem
 Tolerance parameters:
```

```
   fnormtol  =       1e-05
   scsteptol =       1e-05


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

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

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

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

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

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


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

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

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

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

Modified Newton
Solution:
  [x1,x2] =  0.500003     3.1416
Final Statistics:
```

```
   nni =     12     nfe   =     13
   nje =      2     nfeD  =     12


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

2.2   A serial Krylov example: kinFoodWeb_kry

We give here an example that illustrates the use of KINSOL with the Krylov method SPGMR, via the SUNLINSOL_SPGMR module, 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 $c = [c_1, c_2, ..., c_{n_s}]^T$, where $n_s = 2n_p$ is the number of species and n_p is the number of predators and of prey, then the PDEs can be written as

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

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

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

The problem coefficients are given by

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

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

and

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

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

   Homogeneous Neumann boundary conditions are imposed and the initial guess is constant in both x and y. For this example, the equations (2) are discretized spatially with standard central finite differences on a 8×8 mesh with $n_s = 6$, giving a system of size 384.

   Among the initial `#include` lines in this case are lines to include `sunlinsol_spgmr.h` and `sundials_math.h`. The first contains constants and function prototypes associated with the SPGMR solver module. The inclusion of `sundials_math.h` is done to access the SUNMAX

and `SUNRabs` macros, and the `SUNRsqrt` function to compute the square root of a `realtype` number.

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` (see §**??**) to create the SPGMR linear solver module, 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` (see §**??**). The `data` pointer passed to `KINSetUserData` is passed to `PrecSetupBD` and `PrecSolveBD` whenever these are called.

Next, `KINSol` is called, the return value is tested for error conditions, and the approximate solution vector is printed via a call to `PrintOutput`. After that, `PrintFinalStats` is called to get and print final statistics, and memory is freed by calls to `N_VDestroy_Serial`, `FreeUserData` and `KINFree`. The statistics printed are the total numbers of nonlinear iterations (`nni`), of `func` evaluations (excluding those for Jv product evaluations) (`nfe`), of `func` evaluations for Jv evaluations (`nfeSG`), of linear (Krylov) iterations (`nli`), of preconditioner evaluations (`npe`), and of preconditioner solves (`nps`). All of these optional outputs and others are described in §**??**.

Mathematically, the dependent variable has three dimensions: species number, x mesh point, and y mesh point. But in NVECTOR_SERIAL, a vector of type `N_Vector` works with a one-dimensional contiguous array of data components. The macro `IJ_Vptr` isolates the translation from three dimensions to one. Its use results in clearer code and makes it easy to change the underlying layout of the three-dimensional data. Here the problem size is 384, so we use the `NV_DATA_S` macro for efficient `N_Vector` access. The `NV_DATA_S` macro gives a pointer to the first component of a serial `N_Vector` which is then passed to the `IJ_Vptr` macro.

The preconditioner used here is the block-diagonal part of the true Newton matrix and is based only on the partial derivatives of the interaction terms f in (3) and hence its diagonal blocks are $n_s \times n_s$ matrices ($n_s = 6$). It is generated and factored in the `PrecSetupBD` routine and backsolved in the `PrecSolveBD` routine. See §**??** 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 `newDenseMat`, `denseGETRF`, and `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 private 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 retreive 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 10 Newton iterations, with an average of about 38 Krylov iterations per Newton iteration.

```
┌──────────────────── kinFoodWeb_kry sample output ────────────────────┐
│                                                                      │
│  Predator-prey test problem --  KINSol (serial version)             │
│                                                                      │
│  Mesh dimensions = 8 X 8                                            │
│  Number of species = 6                                             │
│  Total system size = 384                                           │
│                                                                      │
│  Flag globalstrategy = 0 (0 = None, 1 = Linesearch)                │
│  Linear solver is SPGMR with maxl = 15, maxlrst = 2                │
│  Preconditioning uses interaction-only block-diagonal matrix       │
│  Positivity constraints imposed on all components                  │
│  Tolerance parameters:  fnormtol = 1e-07   scsteptol = 1e-13       │
│                                                                      │
│  Initial profile of concentration                                  │
│  At all mesh points:  1 1 1   30000 30000 30000                    │
│                                                                      │
│                                                                      │
│  Computed equilibrium species concentrations:                      │
│                                                                      │
│  At bottom left:                                                   │
│   1.16428 1.16428 1.16428 34927.5 34927.5 34927.5                 │
│                                                                      │
│  At top right:                                                    │
│   1.25797 1.25797 1.25797 37736.7 37736.7 37736.7                 │
│                                                                      │
│  Final Statistics..                                               │
│  nni    =      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 module.

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

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

KINBBDPRE uses a band-block-diagonal preconditioner, generated by difference quotients.

The upper and lower half-bandwidths of the Jacobian block generated on each process are both equal to $2 \cdot n_s - 1$, and that is the value 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 $n_s \cdot$ `MXSUB`. However, an even narrower band matrix is retained as the preconditioner, with half-bandwidths equal to n_s, and this is the value passed to `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 aproximation of f used in building the band-block-diagonal preconditioner is already done for the evaluation of f in `func`, a `NULL` pointer is passed as the `gcomm` argument to `KINBBDPrecInit`.

The `main` program resembles closely that of the `kinFoodWeb_kry` example, with particularization arising from the use of the parallel MPI NVECTOR_PARALLEL module. It begins by initializing MPI and obtaining the total number of processes and the rank of the local process. The local length of the solution vector is then computed as NUM_SPECIES·MXSUB·MYSUB. Distributed vectors are created by calling the constructor defined in NVECTOR_PARALLEL with the MPI communicator and the local and global problem sizes as arguments. All output is performed only from the process with id equal to 0. Finally, after `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 51.6 Krylov iterations per Newton iteration.

```
─────────────────── kinFoodWeb_kry_bbd_p sample output ───────────────────

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

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

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

 Flag globalstrategy = 0 (0 = None , 1 = Linesearch)
 Linear solver is SPGMR with maxl = 20, maxlrst = 2
 Preconditioning uses band - block - diagonal matrix from KINBBDPRE
   Difference quotient half - bandwidths: 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
```

3   Fortran example problems

The FORTRAN example problem programs supplied with the KINSOL package are all written in standard F77 Fortran and use double precision arithmetic. Before running any of these examples, the user should make sure that the FORTRAN data types for real and integer variables appropriately match the C types. See §**??** in the KINSOL User Document for details.

However, when the FORTRAN examples are built, the source code is automatically modified according to the configure options supplied by the user and the system type. Integer variables are declared as `INTEGER*n`, where n denotes the number of bytes in the corresponding C type (`long int` or `int`). Floating-point variable declarations remain unchanged if double precision is used, but are changed to `REAL*n`, where n denotes the number of bytes in the SUNDIALS type `realtype`, if using single precision. Also, if using single precision, declarations of floating-point constants are appropriately modified, e.g. `0.5D-4` is changed to `0.5E-4`.

The two examples supplied with the FKINSOL module are very simple tests of the FORTRAN-C interface module. They solve the nonlinear system

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

3.1   A serial example: fkinDiagon_kry

The `fkinDiagon_kry` program solves the above problem using the NVECTOR_SERIAL module.

The main program begins by calling `fnvinits` to initialize computations with the NVECTOR_SERIAL module. Next, the array `uu` is set to contain the initial guess $u_i = 2i$, the array `scale` is set with all components equal to 1.0 (meaning that no scaling is done), and the array `constr` is set with all components equal to 0.0 to indicate that no inequality constraints should be imposed on the solution vector.

The KINSOL solver is initialized and memory for it is allocated by calling `fkinmalloc`, which also specifies the `iout` and `rout` arrays which are used to store integer and real outputs, respectively (see Table **??**). Also, various integer, real, and vector parameters are specified by calling the `fkinsetiin`, `fkinsetrin`, and `fkinsetvin` subroutines, respectively. In particular, the maximum number of iterations between calls to the preconditioner setup routine (`msbpre = 5`), the tolerance for stopping based on the function norm (`fnormtol = 10`$^{-5}$), and the tolerance for stopping based on the step length (`scsteptol = 10`$^{-4}$) are specified.

Next, the SUNLINSOL_SPGMR linear solver module is attached to KINSOL by calling `fsunspgmrinit`, which also specifies the maximum Krylov subspace dimension (`maxl = 10`). This is then attached to KINSOL by calling `fkinlsinit`. The maximum number of restarts allowed for SPGMR is then updated to `maxlrst = 2` by calling `fsunspgmrsetmaxrs`. The SUNLINSOL_SPGMR module is then directed to use the supplied preconditioner by calling the `fkinlssetprec` routine with a first argument equal to 1. The solution of the nonlinear system is obtained after a successful return from `fkinsol`, which is then printed to unit 6 (stdout). Finally, memory allocated for the KINSOL solver is released by calling `fkinfree`.

The user-supplied routine `fkfun` contains a straightforward transcription of the nonlinear system function f, while the routine `fkpset` sets the array `pp` (in the common block `pcom`) to contain an approximation to the reciprocals of the Jacobian diagonal elements. The components of `pp` are then used in `fkpsol` to solve the preconditioner linear system $Px = v$ through simple multiplications.

The following is sample output from `fkinDiagon_kry`, using $N = 128$.

```
┌─────────────────────── fkinDiagon_kry sample output ───────────────────────┐
│                                                                            │
│ Example program fkinDiagon_kry:                                            │
│                                                                            │
│  This FKINSOL example solves a 128 eqn diagonal algebraic system.          │
│  Its purpose is to demonstrate the use of the Fortran interface            │
│  in a serial environment.                                                  │
│                                                                            │
│                                                                            │
│  globalstrategy = KIN_NONE                                                 │
│                                                                            │
│  FKINSOL return code is   0                                                │
│                                                                            │
│                                                                            │
│  The resultant values of uu are:                                          │
│                                                                            │
│    1    1.000000    2.000000    3.000000    4.000000                       │
│    5    5.000000    6.000000    7.000000    8.000000                       │
│    9    9.000000   10.000000   11.000000   12.000000                       │
│   13   13.000000   14.000000   15.000000   16.000000                       │
│   17   17.000000   18.000000   19.000000   20.000000                       │
│   21   21.000000   22.000000   23.000000   24.000000                       │
│   25   25.000000   26.000000   27.000000   28.000000                       │
│   29   29.000000   30.000000   31.000000   32.000000                       │
│   33   33.000000   34.000000   35.000000   36.000000                       │
│   37   37.000000   38.000000   39.000000   40.000000                       │
│   41   41.000000   42.000000   43.000000   44.000000                       │
│   45   45.000000   46.000000   47.000000   48.000000                       │
│   49   49.000000   50.000000   51.000000   52.000000                       │
│   53   53.000000   54.000000   55.000000   56.000000                       │
│   57   57.000000   58.000000   59.000000   60.000000                       │
│   61   61.000000   62.000000   63.000000   64.000000                       │
│   65   65.000000   66.000000   67.000000   68.000000                       │
│   69   69.000000   70.000000   71.000000   72.000000                       │
│   73   73.000000   74.000000   75.000000   76.000000                       │
│   77   77.000000   78.000000   79.000000   80.000000                       │
│   81   81.000000   82.000000   83.000000   84.000000                       │
│   85   85.000000   86.000000   87.000000   88.000000                       │
│   89   89.000000   90.000000   91.000000   92.000000                       │
│   93   93.000000   94.000000   95.000000   96.000000                       │
│   97   97.000000   98.000000   99.000000  100.000000                       │
│  101  101.000000  102.000000  103.000000  104.000000                       │
│  105  105.000000  106.000000  107.000000  108.000000                       │
│  109  109.000000  110.000000  111.000000  112.000000                       │
│  113  113.000000  114.000000  115.000000  116.000000                       │
│  117  117.000000  118.000000  119.000000  120.000000                       │
│  121  121.000000  122.000000  123.000000  124.000000                       │
│  125  125.000000  126.000000  127.000000  128.000000                       │
│                                                                            │
│                                                                            │
│ Final statistics:                                                          │
│                                                                            │
│   nni =    7,  nli  =   21                                                 │
│   nfe =    8,  npe  =    2                                                 │
│   nps =   28,  ncfl =    0                                                 │
│                                                                            │
└────────────────────────────────────────────────────────────────────────────┘
```

3.2 A parallel example: fkinDiagon_kry_p

The program `fkinDiagon_kry_p` is a straightforward modification of `fkinDiagon_kry` to use the MPI-enabled NVECTOR_PARALLEL module.

After initialization of MPI, the NVECTOR_PARALLEL module is initialized by calling `fnvinitp` with the default MPI communicator `mpi_comm_world` and the local and global vector sizes as its first three arguments. The rank of the local process, `mype`, 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 the equation `iglobal = ilocal + mype*nlocal`. In other respects, the problem setup (KINSOL initialization, SUNLINSOL_SPGMR specification) and solution steps are the same as in `fkinDiagon_kry`. Upon successful return from `fkinsol`, the solution segment local to the process with id equal to 0 is printed to unit 6. Finally, the KINSOL memory is released and the MPI environent is terminated.

For this simple example, no inter-process communication is required to evaluate the nonlinear system function f or the preconditioner. As a consequence, the user-supplied routines `fkfun`, `fkpset`, and `fkpsol` are basically identical to those in `fkinDiagon_kry`.

Sample output from `fkinDiagon_kry_p`, for $N = 128$, follows.

```
──────────────── fkinDiagon_kry_p sample output ────────────────

 Example program fkinDiagon_kry_p:

  This FKINSOL example solves a 128 eqn diagonal algebraic system.
  Its purpose is to demonstrate the use of the Fortran interface
  in a parallel environment.

  FKINSOL return code is     0

  The resultant values of uu (process 0) are:

     1    1.000000    2.000000    3.000000    4.000000
     5    5.000000    6.000000    7.000000    8.000000
     9    9.000000   10.000000   11.000000   12.000000
    13   13.000000   14.000000   15.000000   16.000000
    17   17.000000   18.000000   19.000000   20.000000
    21   21.000000   22.000000   23.000000   24.000000
    25   25.000000   26.000000   27.000000   28.000000
    29   29.000000   30.000000   31.000000   32.000000

 Final statistics:

   nni =    7,  nli  =   21
   nfe =    8,  npe  =    2
   nps =   28,  ncfl =    0
```

4 C++ example problems

4.1 A parallel matrix-free example: kin_heat2D_nonlin_p

As an illustration of the use of the KINSOL package for the solution of nonlinear systems in parallel, we give a sample program called `kin_heat2D_nonlin_p.cpp`. It uses the KINSOL KINFP iteration with Anderson Acceleration and the NVECTOR_PARALLEL module (which provides a parallel implementation of NVECTOR) for the solution of the following test problem. 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`).

This problem involves solving a steady-state 2D heat equation with an additional nonlinear term defined by $c(u)$:

$$b = \nabla \cdot (D\nabla u) + c(u) \quad \text{in} \quad \mathcal{D} = [0,1] \times [0,1] \tag{4}$$

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

$$u(0,y) = u(1,y) = u(x,0) = u(x,1) = 0. \tag{5}$$

We chose the analytical solution to be

$$u_{\text{exact}} = u(x,y) = \sin^2(\pi x)\sin^2(\pi y), \tag{6}$$

hence, we define the static term b as follows

$$\begin{aligned} b =& k_x 2\pi^2(\cos^2(\pi x) - \sin^2(\pi x))\sin^2(\pi y) \\ &+ k_y 2\pi^2(\cos^2(\pi y) - \sin^2(\pi y))\sin^2(\pi x) + c(u_{\text{exact}}) \end{aligned} \tag{7}$$

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.,

$$b + u = \nabla \cdot (D\nabla u) + c(u) + u, \tag{8}$$

so that the fixed point function is

$$G(u) = \nabla \cdot (D\nabla u) + c(u) + u - b. \tag{9}$$

The problem is run using a tolerance of 10^{-8}, and a starting vector containing all ones. The following tables contains all available input parameters when running the example problem.

Table 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

Table 2: Input parameter flags for setting the 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`	$c(u) = u + ue^u + ue^{-u}$
`--c 15`	$c(u) = u + ue^u + ue^{-u} + (u - e^u)^2$
`--c 16`	$c(u) = u + ue^u + ue^{-u} + (u - e^u)^2 + (u + u\sin(u) - \cos(u))^2$
`--c 17`	$c(u) = u + ue^{-u} + e^u(u + \sin(u) - \cos(u))^3$

4.2 A parallel example using *hypre*: kin_heat2D_nonlin_hypre_pfmg

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_pfmg.cpp. It uses the KINSOL KINFP iteration and the NVECTOR_PARALLEL module (which provides a parallel implementation of NVECTOR) for the solution of the following test problem. 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).

This problem involves solving a steady-state 2D heat equation with an additional nonlinear term defined by $c(u)$:

$$b = \nabla \cdot (D\nabla u) + c(u) \quad \text{in} \quad \mathcal{D} = [0,1] \times [0,1] \tag{10}$$

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

$$u(0,y) = u(1,y) = u(x,0) = u(x,1) = 0 \tag{11}$$

We chose the analytical solution to be

$$u_{\text{exact}} = u(x,y) = \sin^2(\pi x)\sin^2(\pi y), \tag{12}$$

hence, we define the static term b as follows

$$\begin{aligned} b =& k_x 2\pi^2(\cos^2(\pi x) - \sin^2(\pi x))\sin^2(\pi y) \\ &+ k_y 2\pi^2(\cos^2(\pi y) - \sin^2(\pi y))\sin^2(\pi x) + c(u_{\text{exact}}) \end{aligned} \tag{13}$$

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 formed by implementing the Laplacian as a matrix-vector product,

$$b = Au + c(u), \tag{14}$$

and solving for u results to get the fixed point function

$$G(u) = A^{-1}(b - c(u)). \tag{15}$$

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 SUNLINSOL_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 within the main file.

All input parameter flags available for Example 4.1 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.

17

Table 3: Optional input parameter flags for setting *hypre* related parameters.

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

References

[1] A. M. Collier, A. C. Hindmarsh, R. Serban, and C.S. Woodward. User Documentation for KINSOL v6.4.1. Technical Report UCRL-SM-208116, LLNL, 2022.

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

[3] D. A. Frank-Kamenetskii and N. Thon. *Diffusion and Heat Exchange in Chemical Kinetics.* Princeton University Press, 1955.