User Documentation for KINSOL v3.0.0 (SUNDIALS v3.0.0)

Aaron M. Collier, Alan C. Hindmarsh, Radu Serban, and Carol S. Woodward
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

October 26, 2017

UCRL-SM-208116

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

Contents

List of Tables

List of Figures

Chapter 1

Introduction

KINSOL is part of a software family called SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic equation Solvers [17]. This suite consists of CVODE, ARKODE, KINSOL, and IDA, and variants of these with sensitivity analysis capabilities.

KINSOL is a general-purpose nonlinear system solver based on Newton-Krylov solver technology. A fixed point iteration is also included with the release of KINSOL v.2.8.0 and higher.

1.1 Historical Background

The first nonlinear solver packages based on Newton-Krylov methods were written in FORTRAN. In particular, the NKSOL package, written at LLNL, was the first Newton-Krylov solver package written for solution of systems arising in the solution of partial differential equations [6]. This FORTRAN code made use of Newton's method to solve the discrete nonlinear systems and applied a preconditioned Krylov linear solver for solution of the Jacobian system at each nonlinear iteration. The key to the Newton-Krylov method was that the matrix-vector multiplies required by the Krylov method could effectively be approximated by a finite difference of the nonlinear system-defining function, avoiding a requirement for the formation of the actual Jacobian matrix. Significantly less memory was required for the solver as a result.

In the late 1990's, there was a push at LLNL to rewrite the nonlinear solver in C and port it to distributed memory parallel machines. Both Newton and Krylov methods are easily implemented in parallel, and this effort gave rise to the KINSOL package. KINSOL is similar to NKSOL in functionality, except that it provides for more options in the choice of linear system methods and tolerances, and has a more modular design to provide flexibility for future enhancements.

At present, KINSOL may utilize a variety of Krylov methods provided in SUNDIALS. These methods include the GMRES (Generalized Minimal RESidual) [26], FGMRES (Flexible Generalized Minimum RESidual) [25], Bi-CGStab (Bi-Conjugate Gradient Stabilized) [27], TFQMR (Transpose-Free Quasi-Minimal Residual) [15], and PCG (Preconditioned Conjugate Gradient) [16] linear iterative methods. As Krylov methods, these require little matrix storage for solving the Newton equations as compared to direct methods. However, the algorithms allow for a user-supplied preconditioner matrix, and, for most problems, preconditioning is essential for an efficient solution. For very large nonlinear algebraic systems, the Krylov methods are preferable over direct linear solver methods, and are often the only feasible choice. Among the Krylov methods in SUNDIALS, we recommend GMRES as the best overall choice. However, users are encouraged to compare all three, especially if encountering convergence failures with GMRES. Bi-CGStab and TFQMR have an advantage in storage requirements, in that the number of workspace vectors they require is fixed, while that number for GMRES depends on the desired Krylov subspace size. FGMRES has an advantage in that it is designed to support preconditioners that vary between iterations (e.g. iterative methods). PCG exhibits rapid convergence and minimal workspace vectors, but only works for symmetric linear systems.

For the sake of completeness in functionality, direct linear system solvers are included in KINSOL. These include methods for both dense and banded linear systems, with Jacobians that are either

user-supplied or generated internally by difference quotients. KINSOL also includes interfaces to the sparse direct solvers KLU [9, 1], and the threaded sparse direct solver, SuperLU_MT [21, 11, 2].

In the process of translating NKSOL into C, the overall KINSOL organization has been changed considerably. One key feature of the KINSOL organization is that a separate module devoted to vector operations was created. This module facilitated extension to multiprosessor environments with minimal impact on the rest of the solver. The vector module design is shared across the SUNDIALS suite. This NVECTOR module is written in terms of abstract vector operations with the actual routines attached by a particular implementation (such as serial or parallel) of NVECTOR. This abstraction allows writing the SUNDIALS solvers in a manner independent of the actual NVECTOR implementation (which can be user-supplied), as well as allowing more than one NVECTOR module linked into an executable file. SUNDIALS (and thus KINSOL) is supplied with serial, MPI-parallel, and both openMP and Pthreads thread-parallel NVECTOR implementations.

There are several motivations for choosing the C language for KINSOL. First, a general movement away from FORTRAN and toward C in scientific computing was apparent. Second, the pointer, structure, and dynamic memory allocation features in C are extremely useful in software of this complexity, with the great variety of method options offered. Finally, we prefer C over C++ for KINSOL because of the wider availability of C compilers, the potentially greater efficiency of C, and the greater ease of interfacing the solver to applications written in FORTRAN.

1.2 Changes from previous versions

Changes in v3.0.0

All interfaces to matrix structures and linear solvers have been reworked, and all example programs have been updated. The goal of the redesign of these interfaces was to provide more encapsulation and ease in the interfacing of custom linear solvers and interoperability with linear solver libraries. Specific changes include:

- Added generic SUNMATRIX module with three provided implementations: dense, banded and sparse. These replicate previous SUNDIALS Dls and Sls matrix structures in a single object-oriented API.
- Added example problems demonstrating use of generic SUNMATRIX modules.
- Added generic `SUNLinearSolver` module with eleven provided implementations: SUNDIALS native dense, SUNDIALS native banded, LAPACK dense, LAPACK band, KLU, SuperLU_MT, SPGMR, SPBCGS, SPTFQMR, SPFGMR, and PCG. These replicate previous SUNDIALS generic linear solvers in a single object-oriented API.
- Added example problems demonstrating use of generic SUNLINEARSOLVER modules.
- Expanded package-provided direct linear solver (Dls) interfaces and scaled, preconditioned, iterative linear solver (Spils) interfaces to utilize generic SUNMATRIX and SUNLINEARSOLVER objects.
- Removed package-specific, linear solver-specific, solver modules (e.g. CVDENSE, KINBAND, IDAKLU, ARKSPGMR) since their functionality is entirely replicated by the generic Dls/Spils interfaces and SUNLINEARSOLVER/SUNMATRIX modules. The exception is CVDIAG, a diagonal approximate Jacobian solver available to CVODE and CVODES.
- Converted all SUNDIALS example problems to utilize new generic SUNMATRIX and SUNLINEARSOLVER objects, along with updated Dls and Spils linear solver interfaces.
- Added Spils interface routines to ARKode, CVODE, CVODES, IDA and IDAS to allow specification of a user-provided "JTSetup" routine. This change supports users who wish to set up data structures for the user-provided Jacobian-times-vector ("JTimes") routine, and where the cost of one JTSetup setup per Newton iteration can be amortized between multiple JTimes calls.

Two additional NVECTOR implementations were added – one for CUDA and one for RAJA vectors. These vectors are supplied to provide very basic support for running on GPU architectures. Users are advised that these vectors both move all data to the GPU device upon construction, and speedup will only be realized if the user also conducts the right-hand-side function evaluation on the device. In addition, these vectors assume the problem fits on one GPU. Further information about RAJA, users are referred to th web site, https://software.llnl.gov/RAJA/. These additions are accompanied by additions to various interface functions and to user documentation.

All indices for data structures were updated to a new `sunindextype` that can be configured to be a 32- or 64-bit integer data index type. `sunindextype` is defined to be `int32_t` or `int64_t` when portable types are supported, otherwise it is defined as `int` or `long int`. The Fortran interfaces continue to use `long int` for indices, except for their sparse matrix interface that now uses the new `sunindextype`. This new flexible capability for index types includes interfaces to PETSc, hypre, SuperLU_MT, and KLU with either 32-bit or 64-bit capabilities depending how the user configures SUNDIALS.

To avoid potential namespace conflicts, the macros defining `booleantype` values `TRUE` and `FALSE` have been changed to `SUNTRUE` and `SUNFALSE` respectively.

Temporary vectors were removed from preconditioner setup and solve routines for all packages. It is assumed that all necessary data for user-provided preconditioner operations will be allocated and stored in user-provided data structures.

The file `include/sundials_fconfig.h` was added. This file contains SUNDIALS type information for use in Fortran programs.

The build system was expanded to support many of the xSDK-compliant keys. The xSDK is a movement in scientific software to provide a foundation for the rapid and efficient production of high-quality, sustainable extreme-scale scientific applications. More information can be found at, https://xsdk.info.

Added functions `SUNDIALSGetVersion` and `SUNDIALSGetVersionNumber` to get SUNDIALS release version information at runtime.

In addition, numerous changes were made to the build system. These include the addition of separate `BLAS_ENABLE` and `BLAS_LIBRARIES` CMake variables, additional error checking during CMake configuration, minor bug fixes, and renaming CMake options to enable/disable examples for greater clarity and an added option to enable/disable Fortran 77 examples. These changes included changing `EXAMPLES_ENABLE` to `EXAMPLES_ENABLE_C`, changing `CXX_ENABLE` to `EXAMPLES_ENABLE_CXX`, changing `F90_ENABLE` to `EXAMPLES_ENABLE_F90`, and adding an `EXAMPLES_ENABLE_F77` option.

A bug fix was done to correct the fcmix name translation for `FKIN_SPFGMR`.

Corrections and additions were made to the examples, to installation-related files, and to the user documentation.

Changes in v2.9.0

Two additional NVECTOR implementations were added – one for Hypre (parallel) vectors, and one for PetSC vectors. These additions are accompanied by additions to various interface functions and to user documentation.

Each NVECTOR module now includes a function, `N_VGetVectorID`, that returns the NVECTOR module name.

The Picard iteration return was chanegd to always return the newest iterate upon success. A minor bug in the line search was fixed to prevent an infinite loop when the beta condition fails and lamba is below the minimum size.

For each linear solver, the various solver performance counters are now initialized to 0 in both the solver specification function and in solver `linit` function. This ensures that these solver counters are initialized upon linear solver instantiation as well as at the beginning of the problem solution.

A memory leak was fixed in the banded preconditioner interface. In addition, updates were done to return integers from linear solver and preconditioner 'free' functions.

Corrections were made to three Fortran interface functions. The Anderson acceleration scheme was enhanced by use of QR updating.

The Krylov linear solver Bi-CGstab was enhanced by removing a redundant dot product. Various additions and corrections were made to the interfaces to the sparse solvers KLU and SuperLU_MT, including support for CSR format when using KLU.

The functions FKINCREATE and FKININIT were added to split the FKINMALLOC routine into two pieces. FKINMALLOC remains for backward compatibility, but documentation for it has been removed.

A new examples was added for use of the openMP vector.

Minor corrections and additions were made to the KINSOL solver, to the Fortran interfaces, to the examples, to installation-related files, and to the user documentation.

Changes in v2.8.0

Two major additions were made to the globalization strategy options (`KINSol` argument `strategy`). One is fixed-point iteration, and the other is Picard iteration. Both can be accelerated by use of the Anderson acceleration method. See the relevant paragraphs in Chapter 2.

Three additions were made to the linear system solvers that are available for use with the KINSOL solver. First, in the serial case, an interface to the sparse direct solver KLU was added. Second, an interface to SuperLU_MT, the multi-threaded version of SuperLU, was added as a thread-parallel sparse direct solver option, to be used with the serial version of the NVECTOR module. As part of these additions, a sparse matrix (CSC format) structure was added to KINSOL. Finally, a variation of GMRES called Flexible GMRES was added.

Otherwise, only relatively minor modifications were made to KINSOL:

In function `KINStop`, two return values were corrected to make the values of `uu` and `fval` consistent.

A bug involving initialization of `mxnewtstep` was fixed. The error affects the case of repeated user calls to `KINSol` with no intervening call to `KINSetMaxNewtonStep`.

A bug in the increments for difference quotient Jacobian approximations was fixed in function `kinDlsBandDQJac`.

In `KINLapackBand`, the line `smu = MIN(N-1,mu+ml)` was changed to `smu = mu + ml` to correct an illegal input error for `DGBTRF/DGBTRS`.

In order to avoid possible name conflicts, the mathematical macro and function names `MIN`, `MAX`, `SQR`, `RAbs`, `RSqrt`, `RExp`, `RPowerI`, and `RPowerR` were changed to `SUNMIN`, `SUNMAX`, `SUNSQR`, `SUNRabs`, `SUNRsqrt`, `SUNRexp`, `SRpowerI`, and `SUNRpowerR`, respectively. These names occur in both the solver and in various example programs.

In the FKINSOL module, an incorrect return value `ier` in `FKINfunc` was fixed.

In the FKINSOL optional input routines `FKINSETIIN`, `FKINSETRIN`, and `FKINSETVIN`, the optional fourth argument `key_length` was removed, with hardcoded key string lengths passed to all `strncmp` tests.

In all FKINSOL examples, integer declarations were revised so that those which must match a C type `long int` are declared `INTEGER*8`, and a comment was added about the type match. All other integer declarations are just `INTEGER`. Corresponding minor corrections were made to the user guide.

Two new NVECTOR modules have been added for thread-parallel computing environments — one for openMP, denoted `NVECTOR_OPENMP`, and one for Pthreads, denoted `NVECTOR_PTHREADS`.

With this version of SUNDIALS, support and documentation of the Autotools mode of installation is being dropped, in favor of the CMake mode, which is considered more widely portable.

Changes in v2.7.0

One significant design change was made with this release: The problem size and its relatives, bandwidth parameters, related internal indices, pivot arrays, and the optional output `lsflag` have all been changed from type `int` to type `long int`, except for the problem size and bandwidths in user calls to routines specifying BLAS/LAPACK routines for the dense/band linear solvers. The function `NewIntArray` is replaced by a pair `NewIntArray`/`NewLintArray`, for `int` and `long int` arrays, respectively.

A large number of errors have been fixed. Three major logic bugs were fixed – involving updating the solution vector, updating the linesearch parameter, and a missing error return. Three minor errors were fixed – involving setting `etachoice` in the Matlab/KINSOL interface, a missing error case in `KINPrintInfo`, and avoiding an exponential overflow in the evaluation of `omega`. In each linear solver interface function, the linear solver memory is freed on an error return, and the `**Free` function now includes a line setting to NULL the main memory pointer to the linear solver memory. In the installation files, we modified the treatment of the macro SUNDIALS_USE_GENERIC_MATH, so that the parameter GENERIC_MATH_LIB is either defined (with no value) or not defined.

Changes in v2.6.0

This release introduces a new linear solver module, based on Blas and Lapack for both dense and banded matrices.

The user interface has been further refined. Some of the API changes involve: (a) a reorganization of all linear solver modules into two families (besides the already present family of scaled preconditioned iterative linear solvers, the direct solvers, including the new Lapack-based ones, were also organized into a *direct* family); (b) maintaining a single pointer to user data, optionally specified through a `Set`-type function; (c) a general streamlining of the band-block-diagonal preconditioner module distributed with the solver.

Changes in v2.5.0

The main changes in this release involve a rearrangement of the entire SUNDIALS source tree (see §3.1). At the user interface level, the main impact is in the mechanism of including SUNDIALS header files which must now include the relative path (e.g. `#include <cvode/cvode.h>`). Additional changes were made to the build system: all exported header files are now installed in separate subdirectories of the installation *include* directory.

The functions in the generic dense linear solver (`sundials_dense` and `sundials_smalldense`) were modified to work for rectangular $m \times n$ matrices ($m \leq n$), while the factorization and solution functions were renamed to `DenseGETRF`/`denGETRF` and `DenseGETRS`/`denGETRS`, respectively. The factorization and solution functions in the generic band linear solver were renamed `BandGBTRF` and `BandGBTRS`, respectively.

Changes in v2.4.0

KINSPBCG, KINSPTFQMR, KINDENSE, and KINBAND modules have been added to interface with the Scaled Preconditioned Bi-CGStab (SPBCGS), Scaled Preconditioned Transpose-Free Quasi-Minimal Residual (SPTFQMR), DENSE, and BAND linear solver modules, respectively. (For details see Chapter 4.) Corresponding additions were made to the FORTRAN interface module FKINSOL. At the same time, function type names for Scaled Preconditioned Iterative Linear Solvers were added for the user-supplied Jacobian-times-vector and preconditioner setup and solve functions.

Regarding the FORTRAN interface module FKINSOL, optional inputs are now set using `FKINSETIIN` (integer inputs), `FKINSETRIN` (real inputs), and `FKINSETVIN` (vector inputs). Optional outputs are still obtained from the `IOUT` and `ROUT` arrays which are owned by the user and passed as arguments to `FKINMALLOC`.

The KINDENSE and KINBAND linear solver modules include support for nonlinear residual monitoring which can be used to control Jacobian updating.

To reduce the possibility of conflicts, the names of all header files have been changed by adding unique prefixes (`kinsol_` and `sundials_`). When using the default installation procedure, the header files are exported under various subdirectories of the target `include` directory. For more details see Appendix A.

Changes in v2.3.0

The user interface has been further refined. Several functions used for setting optional inputs were combined into a single one. Additionally, to resolve potential variable scope issues, all SUNDIALS solvers release user data right after its use. The build system has been further improved to make it more robust.

Changes in v2.2.1

The changes in this minor SUNDIALS release affect only the build system.

Changes in v2.2.0

The major changes from the previous version involve a redesign of the user interface across the entire SUNDIALS suite. We have eliminated the mechanism of providing optional inputs and extracting optional statistics from the solver through the `iopt` and `ropt` arrays. Instead, KINSOL now provides a set of routines (with prefix `KINSet`) to change the default values for various quantities controlling the solver and a set of extraction routines (with prefix `KINGet`) to extract statistics after return from the main solver routine. Similarly, each linear solver module provides its own set of `Set`- and `Get`-type routines. For more details see Chapter 4.

Additionally, the interfaces to several user-supplied routines (such as those providing Jacobian-vector products and preconditioner information) were simplified by reducing the number of arguments. The same information that was previously accessible through such arguments can now be obtained through `Get`-type functions.

Installation of KINSOL (and all of SUNDIALS) has been completely redesigned and is now based on configure scripts.

1.3 Reading this User Guide

This user guide is a combination of general usage instructions and specific examples. We expect that some readers will want to concentrate on the general instructions, while others will refer mostly to the examples, and the organization is intended to accommodate both styles.

There are different possible levels of usage of KINSOL. The most casual user, with a small nonlinear system, can get by with reading all of Chapter 2, then Chapter 4 through §4.5.3 only, and looking at examples in [8]. In a different direction, a more expert user with a nonlinear system may want to (a) use a package preconditioner (§4.7), (b) supply his/her own Jacobian or preconditioner routines (§4.6), (c) supply a new NVECTOR module (Chapter 6), or even (d) supply a different linear solver module (§3.2 and Chapter 8).

The structure of this document is as follows:

- In Chapter 2, we provide short descriptions of the numerical methods implemented by KINSOL for the solution of nonlinear systems.
- The following chapter describes the structure of the SUNDIALS suite of solvers (§3.1) and the software organization of the KINSOL solver (§3.2).
- Chapter 4 is the main usage document for KINSOL for C applications. It includes a complete description of the user interface for the solution of nonlinear algebraic systems.
- In Chapter 5, we describe FKINSOL, an interface module for the use of KINSOL with FORTRAN applications.
- Chapter 6 gives a brief overview of the generic NVECTOR module shared among the various components of SUNDIALS, and details on the four NVECTOR implementations provided with SUNDIALS.

- Chapter 7 gives a brief overview of the generic SUNMATRIX module shared among the various components of SUNDIALS, and details on the SUNMATRIX implementations provided with SUNDIALS: a dense implementation (§7.1), a banded implementation (§7.2) and a sparse implementation (§7.3).
- Chapter 8 gives a brief overview of the generic SUNLINSOL module shared among the various components of SUNDIALS. This chapter contains details on the SUNLINSOL implementations provided with SUNDIALS. The chapter also contains details on the SUNLINSOL implementations provided with SUNDIALS that interface with external linear solver libraries.
- Finally, in the appendices, we provide detailed instructions for the installation of KINSOL, within the structure of SUNDIALS (Appendix A), as well as a list of all the constants used for input to and output from KINSOL functions (Appendix B).

Finally, the reader should be aware of the following notational conventions in this user guide: program listings and identifiers (such as `KINInit`) within textual explanations appear in typewriter type style; fields in C structures (such as *content*) appear in italics; and packages or modules are written in all capitals. Usage and installation instructions that constitute important warnings are marked with a triangular symbol in the margin.

Acknowledgments. We wish to acknowledge the contributions to previous versions of the KINSOL code and user guide by Allan G. Taylor.

1.4 SUNDIALS Release License

The SUNDIALS packages are released open source, under a BSD license. The only requirements of the BSD license are preservation of copyright and a standard disclaimer of liability. Our Copyright notice is below along with the license.

PLEASE NOTE If you are using SUNDIALS with any third party libraries linked in (e.g., LaPACK, KLU, SuperLU_MT, PETSc, or *hypre*), be sure to review the respective license of the package as that license may have more restrictive terms than the SUNDIALS license. For example, if someone builds SUNDIALS with a statically linked KLU, the build is subject to terms of the LGPL license (which is what KLU is released with) and *not* the SUNDIALS BSD license anymore.

1.4.1 Copyright Notices

All SUNDIALS packages except ARKode are subject to the following Copyright notice.

1.4.1.1 SUNDIALS Copyright

1.4.1.2 ARKode Copyright

1.4.2 BSD License

Chapter 2

Mathematical Considerations

KINSOL solves nonlinear algebraic systems in real N-space.

Using Newton's method, or the Picard iteration, one can solve

$$F(u) = 0\,, \quad F : \mathbf{R}^N \to \mathbf{R}^N\,, \tag{2.1}$$

given an initial guess u_0. Using a fixed-point iteration, the convergence of which can be improved with Anderson acceleration, one can solve

$$G(u) = u\,, \quad G : \mathbf{R}^N \to \mathbf{R}^N\,, \tag{2.2}$$

given an initial guess u_0.

Basic Newton iteration

Depending on the linear solver used, KINSOL can employ either an Inexact Newton method [4, 6, 10, 12, 20], or a Modified Newton method. At the highest level, KINSOL implements the following iteration scheme:

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 $u_{n+1} = u_n + \lambda\delta_n$, $0 < \lambda \leq 1$
   (c) Test for convergence

Here, u_n is the nth iterate to u, and $J(u) = F'(u)$ is the system Jacobian. At each stage in the iteration process, a scalar multiple of the step δ_n, is added to u_n to produce a new iterate, u_{n+1}. A test for convergence is made before the iteration continues.

Newton method variants

For solving the linear system given in step 2(a), KINSOL provides several choices, including the option of a user-supplied linear solver module. The linear solver modules distributed with SUNDIALS are organized in two families, a *direct* family comprising direct linear solvers for dense, banded, or sparse matrices and a *spils* family comprising scaled preconditioned iterative (Krylov) linear solvers. The methods offered through these modules are as follows:

- dense direct solvers, using either an internal implementation or a Blas/Lapack implementation (serial or threaded vector modules only),

- band direct solvers, using either an internal implementation or a Blas/Lapack implementation (serial or threaded vector modules only),
- sparse direct solver interfaces, using either the KLU sparse solver library [9, 1], or the thread-enabled SuperLU_MT sparse solver library [21, 11, 2] (serial or threaded vector modules only) [Note that users will need to download and install the KLU or SUPERLUMT packages independent of KINSOL],
- SPGMR, a scaled preconditioned GMRES (Generalized Minimal Residual method) solver,
- SPFGMR, a scaled preconditioned FGMRES (Flexible Generalized Minimal Residual method) solver,
- SPBCGS, a scaled preconditioned Bi-CGStab (Bi-Conjugate Gradient Stable method) solver,
- SPTFQMR, a scaled preconditioned TFQMR (Transpose-Free Quasi-Minimal Residual method) solver, or
- PCG, a scaled preconditioned CG (Conjugate Gradient method) solver.

When using one of the direct linear solvers, the linear system in 2(a) is solved exactly, thus resulting in a Modified Newton method (the Jacobian matrix is normally out of date; see below[1]). Note that the dense, band, and sparse direct linear solvers can only be used with the serial and threaded vector representations.

On the other hand, when using any of the iterative linear solvers (GMRES, FGMRES, Bi-CGStab, TFQMR, CG), the linear system in 2(a) is solved only approximately, thus resulting in an Inexact Newton method. Here right preconditioning is available by way of the preconditioning setup and solve routines supplied by the user, in which case the iterative method is applied to the linear systems $(JP^{-1})(P\delta) = -F$, where P denotes the right preconditioning matrix.

Jacobian information update strategy

In general, unless specified otherwise by the user, KINSOL strives to update Jacobian information (the actual system Jacobian J in the case of direct linear solvers, or the preconditioner matrix P in the case of iterative linear solvers) as infrequently as possible to balance the high costs of matrix operations against other costs. Specifically, these updates occur when:

- the problem is initialized,
- $\|\lambda\delta_{n-1}\|_{D_u,\infty} > 1.5$ (Inexact Newton only),
- `mbset`$= 10$ nonlinear iterations have passed since the last update,
- the linear solver failed recoverably with outdated Jacobian information,
- the global strategy failed with outdated Jacobian information, or
- $\|\lambda\delta_n\|_{D_u,\infty} <$ STEPTOL with outdated Jacobian information.

KINSOL allows, through optional solver inputs, changes to the above strategy. Indeed, the user can disable the initial Jacobian information evaluation or change the default value of `mbset`, the number of nonlinear iterations after which a Jacobian information update is enforced.

[1] KINSOL allows the user to enforce a Jacobian evaluation at each iteration thus allowing for an Exact Newton iteration.

Scaling

To address the case of ill-conditioned nonlinear systems, KINSOL allows prescribing scaling factors both for the solution vector and for the residual vector. For scaling to be used, the user should supply values D_u, which are diagonal elements of the scaling matrix such that $D_u u_n$ has all components roughly the same magnitude when u_n is close to a solution, and D_F, which are diagonal scaling matrix elements such that $D_F F$ has all components roughly the same magnitude when u_n is not too close to a solution. In the text below, we use the following scaled norms:

$$\|z\|_{D_u} = \|D_u z\|_2, \quad \|z\|_{D_F} = \|D_F z\|_2, \quad \|z\|_{D_u,\infty} = \|D_u z\|_\infty, \quad \text{and} \quad \|z\|_{D_F,\infty} = \|D_F z\|_\infty \tag{2.3}$$

where $\|\cdot\|_\infty$ is the max norm. When scaling values are provided for the solution vector, these values are automatically incorporated into the calculation of the perturbations used for the default difference quotient approximations for Jacobian information; see (2.7) and (2.9) below.

Globalization strategy

Two methods of applying a computed step δ_n to the previously computed solution vector are implemented. The first and simplest is the standard Newton strategy which applies step 2(b) as above with λ always set to 1. The other method is a global strategy, which attempts to use the direction implied by δ_n in the most efficient way for furthering convergence of the nonlinear problem. This technique is implemented in the second strategy, called Linesearch. This option employs both the α and β conditions of the Goldstein-Armijo linesearch given in [12] for step 2(b), where λ is chosen to guarantee a sufficient decrease in F relative to the step length as well as a minimum step length relative to the initial rate of decrease of F. One property of the algorithm is that the full Newton step tends to be taken close to the solution.

KINSOL implements a backtracking algorithm to first find the value λ such that $u_n + \lambda\delta_n$ satisfies the sufficient decrease condition (or α-condition)

$$F(u_n + \lambda\delta_n) \leq F(u_n) + \alpha\nabla F(u_n)^T\lambda\delta_n\,,$$

where $\alpha = 10^{-4}$. Although backtracking in itself guarantees that the step is not too small, KINSOL secondly relaxes λ to satisfy the so-called β-condition (equivalent to Wolfe's curvature condition):

$$F(u_n + \lambda\delta_n) \geq F(u_n) + \beta\nabla F(u_n)^T\lambda\delta_n\,,$$

where $\beta = 0.9$. During this second phase, λ is allowed to vary in the interval $[\lambda_{min}, \lambda_{max}]$ where

$$\lambda_{min} = \frac{\text{STEPTOL}}{\|\bar{\delta}_n\|_\infty}\,, \quad \bar{\delta}_n^j = \frac{\delta_n^j}{1/D_u^j + |u^j|}\,,$$

and λ_{max} corresponds to the maximum feasible step size at the current iteration (typically $\lambda_{max} = \text{STEPMAX}/\|\delta_n\|_{D_u}$). In the above expressions, v^j denotes the jth component of a vector v.

For more details, the reader is referred to [12].

Nonlinear iteration stopping criteria

Stopping criteria for the Newton method are applied to both of the nonlinear residual and the step length. For the former, the Newton iteration must pass a stopping test

$$\|F(u_n)\|_{D_F,\infty} < \text{FTOL}\,,$$

where FTOL is an input scalar tolerance with a default value of $U^{1/3}$. Here U is the machine unit roundoff. For the latter, the Newton method will terminate when the maximum scaled step is below a given tolerance

$$\|\lambda\delta_n\|_{D_u,\infty} < \text{STEPTOL}\,,$$

where STEPTOL is an input scalar tolerance with a default value of $U^{2/3}$. Only the first condition (small residual) is considered a successful completion of KINSOL. The second condition (small step) may indicate that the iteration is stalled near a point for which the residual is still unacceptable.

Additional constraints

As a user option, KINSOL permits the application of inequality constraints, $u^i > 0$ and $u^i < 0$, as well as $u^i \geq 0$ and $u^i \leq 0$, where u^i is the ith component of u. Any such constraint, or no constraint, may be imposed on each component. KINSOL will reduce step lengths in order to ensure that no constraint is violated. Specifically, if a new Newton iterate will violate a constraint, the maximum step length along the Newton direction that will satisfy all constraints is found, and δ_n in Step 2(b) is scaled to take a step of that length.

Residual monitoring for Modified Newton method

When using a Modified Newton method (i.e. when a direct linear solver is used), in addition to the strategy described above for the update of the Jacobian matrix, KINSOL also provides an optional nonlinear residual monitoring scheme to control when the system Jacobian is updated. Specifically, a Jacobian update will also occur when `mbsetsub`$= 5$ nonlinear iterations have passed since the last update and

$$\|F(u_n)\|_{D_F} > \omega \|F(u_m)\|_{D_F} \,,$$

where u_n is the current iterate and u_m is the iterate at the last Jacobian update. The scalar ω is given by

$$\omega = \min\left(\omega_{min}\, e^{\max(0,\rho-1)}, \omega_{max}\right) \,, \tag{2.4}$$

with ρ defined as

$$\rho = \frac{\|F(u_n)\|_{D_F}}{\text{FTOL}} \,, \tag{2.5}$$

where FTOL is the input scalar tolerance discussed before. Optionally, a constant value ω_{const} can be used for the parameter ω.

The constants controlling the nonlinear residual monitoring algorithm can be changed from their default values through optional inputs to KINSOL. These include the parameters ω_{min} and ω_{max}, the constant value ω_{const}, and the threshold `mbsetsub`.

Stopping criteria for iterative linear solvers

When using an Inexact Newton method (i.e. when an iterative linear solver is used), the convergence of the overall nonlinear solver is intimately coupled with the accuracy with which the linear solver in 2(a) above is solved. KINSOL provides three options for stopping criteria for the linear system solver, including the two algorithms of Eisenstat and Walker [13]. More precisely, the Krylov iteration must pass a stopping test

$$\|J\delta_n + F\|_{D_F} < (\eta_n + U)\|F\|_{D_F} \,,$$

where η_n is one of:

Eisenstat and Walker Choice 1

$$\eta_n = \frac{|\ \|F(u_n)\|_{D_F} - \|F(u_{n-1}) + J(u_{n-1})\delta_n\|_{D_F}\ |}{\|F(u_{n-1})\|_{D_F}} \,,$$

Eisenstat and Walker Choice 2

$$\eta_n = \gamma \left(\frac{\|F(u_n)\|_{D_F}}{\|F(u_{n-1})\|_{D_F}}\right)^{\alpha} \,,$$

where default values of γ and α are 0.9 and 2, respectively.

Constant η

$$\eta_n = constant,$$

with 0.1 as the default.

The default strategy is "Eisenstat and Walker Choice 1". For both options 1 and 2, appropriate safeguards are incorporated to ensure that η does not decrease too quickly [13].

Difference quotient Jacobian approximations

With the direct linear solver interface, the Jacobian may be supplied by a user routine, or approximated by difference quotients, at the user's option. In the latter case, we use the usual approximation

$$J^{ij} = [F^i(u + \sigma_j e^j) - F^i(u)]/\sigma_j \,. \tag{2.6}$$

The increments σ_j are given by

$$\sigma_j = \sqrt{U}\ \max\left\{|u^j|, 1/D_u^j\right\} \,. \tag{2.7}$$

In the dense case, this scheme requires N evaluations of F, one for each column of J. In the band case, the columns of J are computed in groups, by the Curtis-Powell-Reid algorithm, with the number of F evaluations equal to the bandwidth. The parameter U above can (optionally) be replaced by a user-specified value, `relfunc`.

We note that with the sparse direct solvers, the Jacobian *must* be supplied by a user routine, i.e. it is not approximated internally within KINSOL.

In the case of a Krylov method, Jacobian information is needed only as matrix-vector products Jv. If a routine for Jv is not supplied, these products are approximated by directional difference quotients as

$$J(u)v \approx [F(u + \sigma v) - F(u)]/\sigma \,, \tag{2.8}$$

where u is the current approximation to a root of (2.1), and σ is a scalar. The choice of σ is taken from [6] and is given by

$$\sigma = \frac{\max\{|u^T v|, u_{typ}^T |v|\}}{\|v\|_2^2} \text{sign}(u^T v)\sqrt{U} \,, \tag{2.9}$$

where u_{typ} is a vector of typical values for the absolute values of the solution (and can be taken to be inverses of the scale factors given for u as described below). This formula is suitable for *scaled* vectors u and v, and so is applied to $D_u u$ and $D_u v$. The parameter U above can (optionally) be replaced by a user-specified value, `relfunc`. Convergence of the Newton method is maintained as long as the value of σ remains appropriately small, as shown in [4].

Basic Fixed Point iteration

The basic fixed-point iteration scheme implemented in KINSOL is given by:

1. Set u_0 = an initial guess
2. For $n = 0, 1, 2, ...$ until convergence do:
   (a) Set $u_{n+1} = G(u_n)$.
   (b) Test for convergence.

Here, u_n is the nth iterate to u. At each stage in the iteration process, function G is applied to the current iterate to produce a new iterate, u_{n+1}. A test for convergence is made before the iteration continues.

For Picard iteration, as implemented in KINSOL, we consider a special form of the nonlinear function F, such that $F(u) = Lu - N(u)$, where L is a constant nonsingular matrix and N is (in general) nonlinear. Then the fixed-point function G is defined as $G(u) = u - L^{-1}F(u)$. The Picard iteration is given by:

1. Set u_0 = an initial guess
2. For $n = 0, 1, 2, ...$ until convergence do:
   (a) Set $u_{n+1} = G(u_n) = u_n - L^{-1}F(u_n)$.
   (b) Test $F(u_{n+1})$ for convergence.

Here, u_n is the nth iterate to u. Within each iteration, the Picard step is computed then added to u_n to produce the new iterate. Next, the nonlinear residual function is evaluated at the new iterate, and convergence is checked. Noting that $L^{-1}N(u) = u - L^{-1}F(u)$, the above iteration can be written in the same form as a Newton iteration except that here, L is in the role of the Jacobian. Within KINSOL, however, we leave this in a fixed-point form as above. For more information, see p. 182 of [23].

Anderson Acceleration

The Picard and fixed point methods can be significantly accelerated using Anderson's method [3, 28, 14, 22]. Anderson acceleration can be formulated as follows:

1. Set $u_0 =$ an initial guess and $m \geq 1$
2. Set $u_1 = G(u_0)$
3. For $n = 0, 1, 2, ...$ until convergence do:
   (a) Set $m_n = \min\{m, n\}$
   (b) Set $F_n = (f_{n-m_n}, \ldots, f_n)$, where $f_i = G(u_i) - u_i$
   (c) Determine $\alpha^{(n)} = (\alpha_0^{(n)}, \ldots, \alpha_{m_n}^{(n)})$ that solves $\min_\alpha \|F_n \alpha^T\|_2$ such that $\sum_{i=0}^{m_n} \alpha_i = 1$
   (d) Set $u_{n+1} = \sum_{i=0}^{m_n} \alpha_i^{(n)} G(u_{n-m_n+i})$
   (e) Test for convergence

It has been implemented in KINSOL by turning the constrained linear least-squares problem in Step (c) into an unconstrained one leading to the algorithm given below:

1. Set $u_0 =$ an initial guess and $m \geq 1$
2. Set $u_1 = G(u_0)$
3. For $n = 0, 1, 2, ...$ until convergence do:
   (a) Set $m_n = \min\{m, n\}$
   (b) Set $\Delta F_n = (\Delta f_{n-m_n}, \ldots, \Delta f_{n-1})$, where $\Delta f_i = f_{i+1} - f_i$ and $f_i = G(u_i) - u_i$
   (c) Determine $\gamma^{(n)} = (\gamma_0^{(n)}, \ldots, \gamma_{m_n-1}^{(n)})$ that solves $\min_\gamma \|f_n - \Delta F_n \gamma^T\|_2$
   (d) Set $u_{n+1} = G(u_n) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta g_{n-m_n+i}$ with $\Delta g_i = G(u_{i+1}) - G(u_i)$
   (e) Test for convergence

The least-squares problem in (c) is solved by applying a QR factorization to $\Delta F_n = Q_n R_n$ and solving $R_n \gamma = Q_n^T f_n$.

Fixed-point - Anderson Acceleration Stopping Criterion

The default stopping criterion is

$$\|G(u_{n+1}) - u_{n+1}\|_{D_F,\infty} < \text{GTOL}\,,$$

where D_F is a user-defined diagonal matrix that can be the identity or a scaling matrix chosen so that the components of $D_F(G(u) - u)$ have roughly the same order of magnitude. Note that when using Anderson acceleration, convergence is checked after the acceleration is applied.

Picard - Anderson Acceleration Stopping Criterion

The default stopping criterion is

$$\|F(u_{n+1})\|_{D_F,\infty} < \text{FTOL}\,,$$

where D_F is a user-defined diagonal matrix that can be the identity or a scaling matrix chosen so that the components of $D_F F(u)$ have roughly the same order of magnitude. Note that when using Anderson acceleration, convergence is checked after the acceleration is applied.

Chapter 3

Code Organization

3.1 SUNDIALS organization

The family of solvers referred to as SUNDIALS consists of the solvers CVODE and ARKODE (for ODE systems), KINSOL (for nonlinear algebraic systems), and IDA (for differential-algebraic systems). In addition, SUNDIALS also includes variants of CVODE and IDA with sensitivity analysis capabilities (using either forward or adjoint methods), called CVODES and IDAS, respectively.

The various solvers of this family share many subordinate modules. For this reason, it is organized as a family, with a directory structure that exploits that sharing (see Fig. 3.1). The following is a list of the solver packages presently available, and the basic functionality of each:

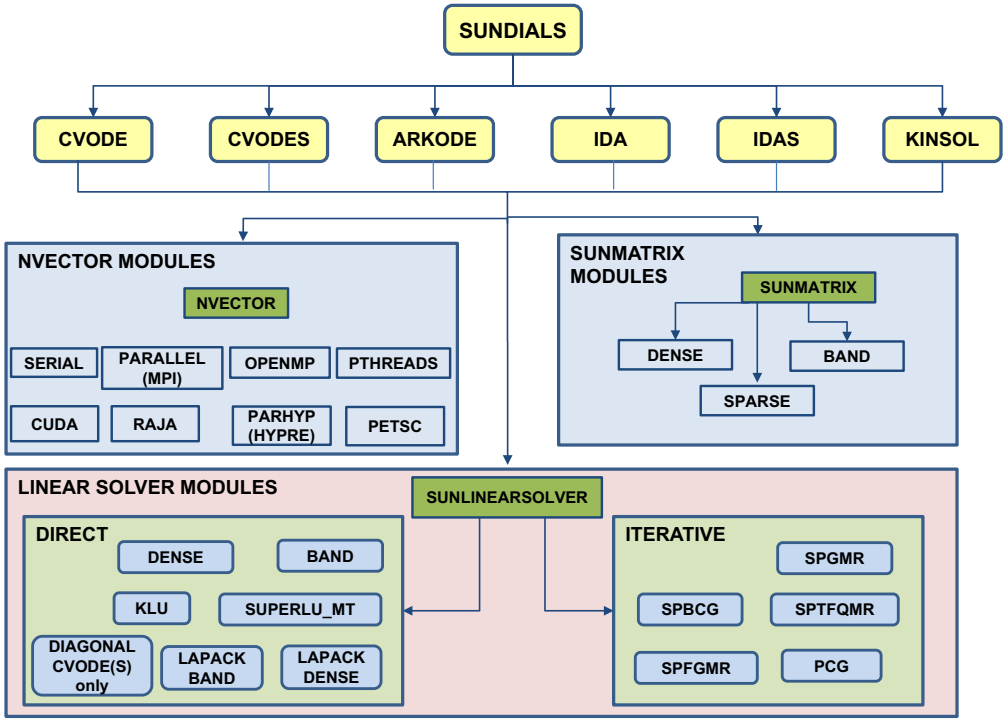
- CVODE, a solver for stiff and nonstiff ODE systems $dy/dt = f(t, y)$ based on Adams and BDF methods;
- CVODES, a solver for stiff and nonstiff ODE systems with sensitivity analysis capabilities;
- ARKODE, a solver for ODE systems $M dy/dt = f_E(t, y) + f_I(t, y)$ based on additive Runge-Kutta methods;
- IDA, a solver for differential-algebraic systems $F(t, y, \dot{y}) = 0$ based on BDF methods;
- IDAS, a solver for differential-algebraic systems with sensitivity analysis capabilities;
- KINSOL, a solver for nonlinear algebraic systems $F(u) = 0$.

3.2 KINSOL organization

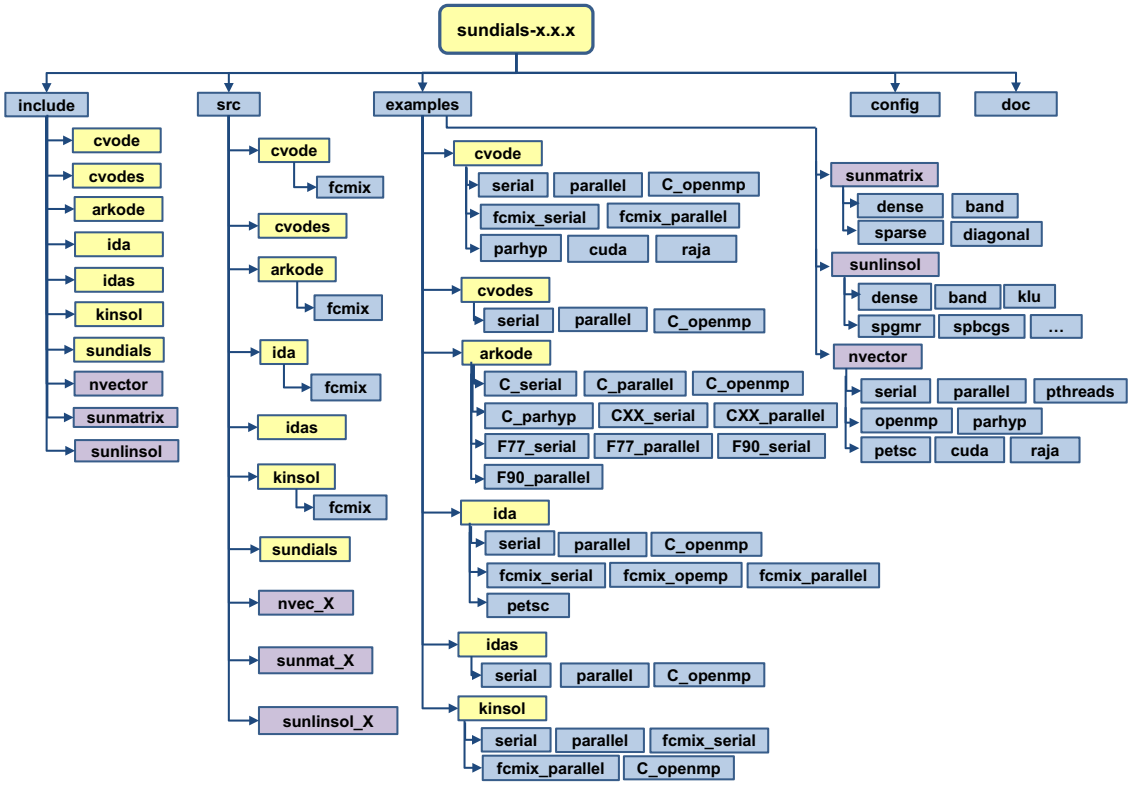
The KINSOL package is written in the ANSI C language. This section summarizes the basic structure of the package, although knowledge of this structure is not necessary for its use.

The overall organization of the KINSOL package is shown in Figure 3.2. The central solver module, implemented in the files `kinsol.h`, `kinsol_impl.h` and `kinsol.c`, deals with the solution of a nonlinear algebraic system using either an Inexact Newton method or a line search method for the global strategy. Although this module contains logic for the Newton iteration, it has no knowledge of the method used to solve the linear systems that arise. For any given user problem, one of the linear system solve rinterfaces is specified, and is then invoked as needed.

At present, the package includes two linear solver interfaces. The *direct* linear solver interface, KINDLS, supports SUNLINSOL implementations with type `SUNLINSOL_DIRECT` (see Chapter 8). These linear solvers utilize direct methods for the solution of linear systems stored using one of the SUNDIALS generic SUNMATRIX implementations (dense, banded or sparse; see Chapter 7). The *spils* linear solver interface, KINSPILS, supports SUNLINSOL implementations with type `SUNLINSOL_ITERATIVE` (see Chapter 8). These linear solvers utilize scaled preconditioned iterative methods. It is assumed that



(a) High-level diagram



(b) Directory structure of the source tree

Figure 3.1: Organization of the SUNDIALS suite

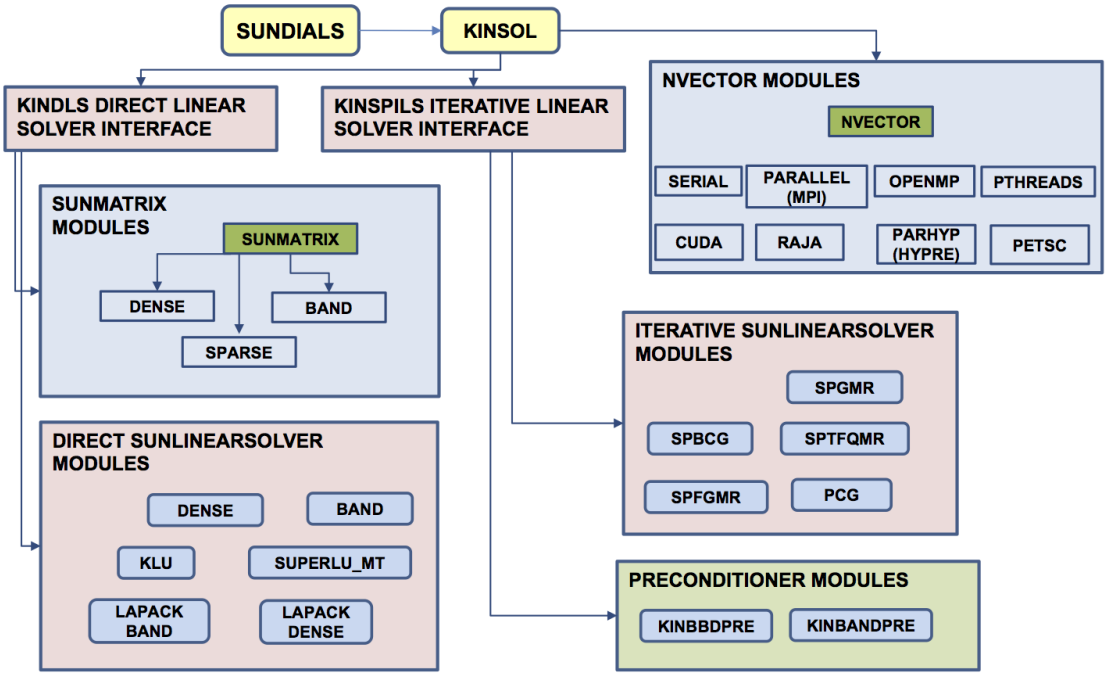


Figure 3.2: Overall structure diagram of the KINSOL package. Modules specific to KINSOL are distinguished by rounded boxes, while generic solver and auxiliary modules are in rectangular boxes. Grayed boxes refer to the encompassing SUNDIALS structure. Note also that the LAPACK, KLU and SUPERLUMT support is through interfaces to external packages. Users will need to download and compile those packages independently.

these methods are implemented in a "matrix-free" manner, wherein only the action of the matrix-vector product is required. Since KINSOL can operate on any valid SUNLINSOL implementation of `SUNLINSOL_DIRECT` or `SUNLINSOL_ITERATIVE` types, the set of linear solver modules available to KINSOL will expand as new SUNLINSOL modules are developed.

Within the KINDLS interface, the package includes algorithms for the approximation of dense or banded Jacobians through difference quotients, but the user also has the option of supplying the Jacobian (or an approximation to it) directly. This user-supplied routine is required when using sparse Jacobian matrices, since standard difference quotient approximations do not leverage the inherent sparsity of the problem.

Within the KINSPILS interface, the package includes an algorithm for the approximation by difference quotients of the product between the Jacobian matrix and a vector. Again, the user has the option of providing routines for this operation, in two phases: setup (preprocessing of Jacobian data) and multiplication. For preconditioned iterative methods, the preconditioning must be supplied by the user, again in two phases: setup and solve. While there is no default choice of preconditioner analogous to the difference-quotient approximation in the direct case, the references [5, 7], together with the example and demonstration programs included with KINSOL, offer considerable assistance in building preconditioners.

Each KINSOL linear solver interface consists of four routines, devoted to (1) memory allocation and initialization, (2) setup of the matrix data involved, (3) solution of the system, and (4) freeing of memory. The setup and solution phases are separate because the evaluation of Jacobians and preconditioners is done only periodically during the solution, as required to achieve convergence. The call list within the central KINSOL module to each of the associated functions is fixed, thus allowing the central module to be completely independent of the linear system method.

These modules are also decomposed in another way. Each of the linear solver modules (KINDENSE, etc.) consists of an interface built on top of a generic linear system solver (DENSE etc.). The interface deals with the use of the particular method in the KINSOL context, whereas the generic solver is independent of the context. While some of the generic linear system solvers (DENSE, BAND, SPGMR, SPFGMR, SPBCGS, SPTFQMR, and PCG) were written with SUNDIALS in mind, they are intended to be usable anywhere as general-purpose solvers. This separation also allows for any generic solver to be replaced by an improved version, with no necessity to revise the KINSOL package elsewhere.

KINSOL also provides a preconditioner module called KINBBDPRE for use with any of the Krylov iterative liear solvers. It works in conjunction with NVECTOR_PARALLEL and generates a preconditioner that is a block-diagonal matrix with each block being a banded matrix, as further described in §4.7.

All state information used by KINSOL to solve a given problem is saved in a structure, and a pointer to that structure is returned to the user. There is no global data in the KINSOL package, and so, in this respect, it is reentrant. State information specific to the linear solver is saved in a separate structure, a pointer to which resides in the KINSOL memory structure. The reentrancy of KINSOL was motivated by the anticipated multi-computer extension.

Chapter 4

Using KINSOL for C Applications

This chapter is concerned with the use of KINSOL for the solution of nonlinear systems. The following subsections treat the header files, the layout of the user's main program, description of the KINSOL user-callable routines, and user-supplied functions. The sample programs described in the companion document [8] may also be helpful. Those codes may be used as templates (with the removal of some lines involved in testing), and are included in the KINSOL package.

Users with applications written in FORTRAN should see Chapter 5, which describes the FORTRAN/C interface module.

The user should be aware that not all SUNLINSOL and SUNMATRIX modules are compatible with all NVECTOR implementations. Details on compatability are given in the documentation for each SUNMATRIX module (Chapter 7) and each SUNLINSOL module (Chapter 8). For example, NVECTOR_PARALLEL is not compatible with the dense, banded, or sparse SUNMATRIX types, or with the corresponding dense, banded, or sparse SUNLINSOL modules. Please check Chapters 7 and 8 to verify compatability between these modules. In addition to that documentation, we note that the preconditioner module KINBBDPRE can only be used with NVECTOR_PARALLEL. It is not recommended to use a threaded vector module with SuperLU_MT unless it is the NVECTOR_OPENMP module, and SuperLU_MT is also compiled with openMP.

KINSOL uses various constants for both input and output. These are defined as needed in this chapter, but for convenience are also listed separately in Appendix B.

4.1 Access to library and header files

At this point, it is assumed that the installation of KINSOL, following the procedure described in Appendix A, has been completed successfully.

Regardless of where the user's application program resides, its associated compilation and load commands must make reference to the appropriate locations for the library and header files required by KINSOL. The relevant library files are

- *libdir*`/libsundials_kinsol.`*lib*,
- *libdir*`/libsundials_nvec*.`*lib* (one to four files),

where the file extension *.lib* is typically `.so` for shared libraries and `.a` for static libraries. The relevant header files are located in the subdirectories

- *incdir*`/include/kinsol`
- *incdir*`/include/sundials`
- *incdir*`/include/sunmatrix`
- *incdir*`/include/sunlinsol`

The directories *libdir* and *incdir* are the install library and include directories, respectively. For a default installation, these are *builddir*/lib and *builddir*/include, respectively, where *builddir* was defined in Appendix A.

4.2 Data types

The sundials_types.h file contains the definition of the type realtype, which is used by the SUNDIALS solvers for all floating-point data, the definition of the integer type sunindextype, which is used for vector and matrix indices, and booleantype, which is used for certain logic operations within SUNDIALS.

4.2.1 Floating point types

The type realtype can be float, double, or long double, with the default being double. The user can change the precision of the SUNDIALS solvers arithmetic at the configuration stage (see §A.1.2).

Additionally, based on the current precision, sundials_types.h defines BIG_REAL to be the largest value representable as a realtype, SMALL_REAL to be the smallest value representable as a realtype, and UNIT_ROUNDOFF to be the difference between 1.0 and the minimum realtype greater than 1.0.

Within SUNDIALS, real constants are set by way of a macro called RCONST. It is this macro that needs the ability to branch on the definition realtype. In ANSI C, a floating-point constant with no suffix is stored as a double. Placing the suffix "F" at the end of a floating point constant makes it a float, whereas using the suffix "L" makes it a long double. For example,

```
#define A 1.0
#define B 1.0F
#define C 1.0L
```

defines A to be a double constant equal to 1.0, B to be a float constant equal to 1.0, and C to be a long double constant equal to 1.0. The macro call RCONST(1.0) automatically expands to 1.0 if realtype is double, to 1.0F if realtype is float, or to 1.0L if realtype is long double. SUNDIALS uses the RCONST macro internally to declare all of its floating-point constants.

A user program which uses the type realtype and the RCONST macro to handle floating-point constants is precision-independent except for any calls to precision-specific standard math library functions. (Our example programs use both realtype and RCONST.) Users can, however, use the type double, float, or long double in their code (assuming that this usage is consistent with the typedef for realtype). Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use realtype, so long as the SUNDIALS libraries use the correct precision (for details see §A.1.2).

4.2.2 Integer types used for vector and matrix indices

The type sunindextype can be either a 32- or 64-bit *signed* integer. The default is the portable int64_t type, and the user can change it to int32_t at the configuration stage. The configuration system will detect if the compiler does not support portable types, and will replace int32_t and int64_t with int and long int, respectively, to ensure use of the desired sizes on Linux, Mac OS X, and Windows platforms. SUNDIALS currently does not support *unsigned* integer types for vector and matrix indices, although these could be added in the future if there is sufficient demand.

A user program which uses sunindextype to handle vector and matrix indices will work with both index storage types except for any calls to index storage-specific external libraries. (Our C and C++ example programs use sunindextype.) Users can, however, use any one of int, long int, int32_t, int64_t or long long int in their code, assuming that this usage is consistent with the typedef for sunindextype on their architecture). Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use sunindextype, so long as the SUNDIALS libraries use the appropriate index storage type (for details see §A.1.2).

4.3 Header files

The calling program must include several header files so that various macros and data types can be used. The header file that is always required is:

- `kinsol/kinsol.h`, the header file for KINSOL, which defines several types and various constants, and includes function prototypes.

`kinsol.h` also includes `sundials_types.h`, which defines the types `realtype`, `sunindextype`, and `booleantype` and constants `SUNFALSE` and `SUNTRUE`.

The calling program must also include an NVECTOR implementation header file, of the form `nvector/nvector_***.h`. See Chapter 6 for the appropriate name. This file in turn includes the header file `sundials_nvector.h` which defines the abstract `N_Vector` data type.

Finally, a linear solver module header file is required. The header files corresponding to the various linear solver interfaces and linear solver modules available for use with KINSOL are:

- `kinsol/kinsol_direct.h`, which is used with the KINDLS direct linear solver interface to access direct solvers with the following header files:
  - `sunlinsol/sunlinsol_dense.h`, which is used with the dense linear solver module, SUNLINSOL_DENSE;
  - `sunlinsol/sunlinsol_band.h`, which is used with the banded linear solver module, SUNLINSOL_BAND;
  - `sunlinsol/sunlinsol_lapackdense.h`, which is used with the LAPACK package dense linear solver interface module, SUNLINSOL_LAPACKDENSE;
  - `sunlinsol/sunlinsol_lapackband.h`, which is used with the LAPACK package banded linear solver interface module, SUNLINSOL_LAPACKBAND;
  - `sunlinsol/sunlinsol_klu.h`, which is used with the KLU sparse linear solver interface module, SUNLINSOL_KLU;
  - `sunlinsol/sunlinsol_superlumt.h`, which is used with the SUPERLUMT sparse linear solver interface module, SUNLINSOL_SUPERLUMT;
- `kinsol/kinsol_spils.h`, which is used with the KINSPILS iterative linear solver interface to access iterative solvers with the following header files:
  - `sunlinsol/sunlinsol_spgmr.h`, which is used with the scaled, preconditioned GMRES Krylov linear solver module, SUNLINSOL_SPGMR;
  - `sunlinsol/sunlinsol_spfgmr.h`, which is used with the scaled, preconditioned FGMRES Krylov linear solver module, SUNLINSOL_SPFGMR;
  - `sunlinsol/sunlinsol_spbcgs.h`, which is used with the scaled, preconditioned Bi-CGStab Krylov linear solver module, SUNLINSOL_SPBCGS;
  - `sunlinsol/sunlinsol_sptfqmr.h`, which is used with the scaled, preconditioned TFQMR Krylov linear solver module, SUNLINSOL_SPTFQMR;
  - `sunlinsol/sunlinsol_pcg.h`, which is used with the scaled, preconditioned CG Krylov linear solver module, SUNLINSOL_PCG;

The header files for the SUNLINSOL_DENSE and SUNLINSOL_LAPACKDENSE linear solver modules include the file `sunmatrix/sunmatrix_dense.h`, which defines the SUNMATRIX_DENSE matrix module, as as well as various functions and macros acting on such matrices.

The header files for the SUNLINSOL_BAND and SUNLINSOL_LAPACKBAND linear solver modules include the file `sunmatrix/sunmatrix_band.h`, which defines the SUNMATRIX_BAND matrix module, as as well as various functions and macros acting on such matrices.

The header files for the SUNLINSOL_KLU and SUNLINSOL_SUPERLUMT sparse linear solvers include the file `sunmatrix/sunmatrix_sparse.h`, which defines the SUNMATRIX_SPARSE matrix module, as well as various functions and macros acting on such matrices.

The header files for the Krylov iterative solvers include the file `sundials/sundials_iterative.h`, which enumerates the kind of preconditioning, and (for the SPGMR and SPFGMR solvers) the choices for the Gram-Schmidt process.

Other headers may be needed, according to the choice of preconditioner, etc. For example, in the `kinFoodWeb_kry_p` example (see [8]), preconditioning is done with a block-diagonal matrix. For this, even though the SUNLINSOL_SPGMR linear solver is used, the header `sundials/sundials_dense.h` is included for access to the underlying generic dense matrix arithmetic routines.

4.4 A skeleton of the user's main program

The following is a skeleton of the user's main program (or calling program) for the solution of a nonlinear system problem. Most of the steps are independent of the NVECTOR, SUNMATRIX, and SUNLINSOL implementations used. For the steps that are not, refer to Chapter 6, 7, and 8 for the specific name of the function to be called or macro to be referenced.

1. **Initialize parallel or multi-threaded environment, if appropriate**

   For example, call `MPI_Init` to initialize MPI if used, or set `num_threads`, the number of threads to use within the threaded vector functions, if used.

2. **Set problem dimensions etc.**

   This generally includes the problem size `N`, and may include the local vector length `Nlocal`.

   Note: The variables `N` and `Nlocal` should be of type `sunindextype`.

3. **Set vector with initial guess**

   To set the vector `u` of initial guess values, use the appropriate functions defined by the particular NVECTOR implementation.

   For native SUNDIALS vector implementations (except the CUDA and RAJA-based ones), use a call of the form `u = N_VMake_***(..., udata)` if the `realtype` array `udata` containing the initial values of u already exists. Otherwise, create a new vector by making a call of the form `u = N_VNew_***(...)`, and then set its elements by accessing the underlying data with a call of the form `ydata = N_VGetArrayPointer(u)`. See §6.1-6.4 for details.

   For the *hypre* and PETSc vector wrappers, first create and initialize the underlying vector and then create an NVECTOR wrapper with a call of the form `u = N_VMake_***(uvec)`, where `uvec` is a *hypre* or PETSc vector. Note that calls like `N_VNew_***(...)` and `N_VGetArrayPointer(...)` are not available for these vector wrappers. See §6.5 and §6.6 for details.

   If using either the CUDA- or RAJA-based vector implementations use a call of the form `u = N_VMake_***(..., c)` where `c` is a pointer to a `suncudavec` or `sunrajavec` vector class if this class already exists. Otherwise, create a new vector by making a call of the form `u = N_VNew_***(...)`, and then set its elements by accessing the underlying data where it is located with a call of the form `N_VGetDeviceArrayPointer_***` or `N_VGetHostArrayPointer_***`. Note that the vector class will allocate memory on both the host and device when instantiated. See §6.7-6.8 for details.

4. **Create KINSOL object**

   Call `kin_mem = KINCreate()` to create the KINSOL memory block. `KINCreate` returns a pointer to the KINSOL memory structure. See §4.5.1 for details.

5. **Set optional inputs**

   Call `KINSet*` routines to change from their default values any optional inputs that control the behavior of KINSOL. See §4.5.4 for details.

6. **Allocate internal memory**

Call `KINInit(...)` to specify the problem defining function F, allocate internal memory for KINSOL, and initialize KINSOL. `KINInit` returns a flag to indicate success or an illegal argument value. See §4.5.1 for details.

7. **Create matrix object**

   If a direct linear solver is to be used within a Newton or Picard iteration, then a template Jacobian matrix must be created by using the appropriate functions defined by the particular SUNMATRIX implementation.

   NOTE: The dense, banded, and sparse matrix objects are usable only in a serial or threaded environment.

8. **Create linear solver object**

   If a Newton or Picard iteration is chosen, then the desired linear solver object must be created by using the appropriate functions defined by the particular SUNLINSOL implementation.

9. **Set linear solver optional inputs**

   Call `*Set*` functions from the selected linear solver module to change optional inputs specific to that linear solver. See the documentation for each SUNLINSOL module in Chapter 8 for details.

10. **Attach linear solver module**

    If a Newton or Picard iteration is chosen, initialize the KINDLS or KINSPILS linear solver interface by attaching the linear solver object (and matrix object, if applicable) with one of the following calls (for details see §4.5.2):

    ```
    ier = KINDlsSetLinearSolver(...);
    ```

    ```
    ier = KINSpilsSetLinearSolver(...);
    ```

11. **Set linear solver interface optional inputs**

    Call `KINDlsSet*` or `KINSpilsSet*` functions to change optional inputs specific to that linear solver interface. See §4.5.4 for details.

12. **Solve problem**

    Call `ier = KINSol(...)` to solve the nonlinear problem for a given initial guess. See §4.5.3 for details.

13. **Get optional outputs**

    Call `KINGet*` and `KIN*Get*` functions to obtain optional output. See §4.5.5 for details.

14. **Deallocate memory for solution vector**

    Upon completion of the solution, deallocate memory for the vector `u` by calling the appropriate destructor function defined by the NVECTOR implementation:

    ```
    N_VDestroy(u);
    ```

15. **Free solver memory**

    Call `KINFree(&kin_mem)` to free the memory allocated for KINSOL.

16. **Free linear solver and matrix memory**

    Call `SUNLinSolFree` and `SUNMatDestroy` to free any memory allocated for the linear solver and matrix objects created above.

17. **Finalize MPI, if used**

    Call `MPI_Finalize()` to terminate MPI.

SUNDIALS provides some linear solvers only as a means for users to get problems running and not as highly efficient solvers. For example, if solving a dense system, we suggest using the Lapack solvers if the size of the linear system is $> 50,000$. (Thanks to A. Nicolai for his testing and recommendation.) Table 4.1 shows the linear solver interfaces available as SUNLINSOL modules and the vector implementations required for use. As an example, one cannot use the dense direct solver interfaces with the MPI-based vector implementation. However, as discussed in Chapter 8 the SUNDIALS packages operate on generic SUNLINSOL objects, allowing a user to develop their own solvers should they so desire.

Table 4.1: SUNDIALS linear solver interfaces and vector implementations that can be used for each.

Linear Solver	Serial	Parallel (MPI)	OpenMP	pThreads	*hypre*	PETSc	CUDA	RAJA	User Supp.
Dense	✓		✓	✓					✓
Band	✓		✓	✓					✓
LapackDense	✓		✓	✓					✓
LapackBand	✓		✓	✓					✓
KLU	✓		✓	✓					✓
SUPERLUMT	✓		✓	✓					✓
SPGMR	✓	✓	✓	✓	✓	✓	✓	✓	✓
SPFGMR	✓	✓	✓	✓	✓	✓	✓	✓	✓
SPBCGS	✓	✓	✓	✓	✓	✓	✓	✓	✓
SPTFQMR	✓	✓	✓	✓	✓	✓	✓	✓	✓
PCG	✓	✓	✓	✓	✓	✓	✓	✓	✓
User Supp.	✓	✓	✓	✓	✓	✓	✓	✓	✓

4.5 User-callable functions

This section describes the KINSOL functions that are called by the user to set up and solve a nonlinear problem. Some of these are required. However, starting with §4.5.4, the functions listed involve optional inputs/outputs or restarting, and those paragraphs can be skipped for a casual use of KINSOL. In any case, refer to §4.4 for the correct order of these calls.

The return flag (when present) for each of these routines is a negative integer if an error occurred, and non-negative otherwise.

4.5.1 KINSOL initialization and deallocation functions

The following three functions must be called in the order listed. The last one is to be called only after the problem solution is complete, as it frees the KINSOL memory block created and allocated by the first two calls.

`KINCreate`

Call `kin_mem = KINCreate();`

Description The function `KINCreate` instantiates a KINSOL solver object.

Arguments This function has no arguments.

Return value If successful, `KINCreate` returns a pointer to the newly created KINSOL memory block (of type `void *`). If an error occurred, `KINCreate` prints an error message to `stderr` and returns `NULL`.

`KINInit`

Call `flag = KINInit(kin_mem, func, tmpl);`

Description The function `KINInit` specifies the problem-defining function, allocates internal memory, and initializes KINSOL.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block returned by `KINCreate`.

`func` (`KINSysFn`) is the C function which computes the system function F (or $G(u)$ for fixed-point iteration) in the nonlinear problem. This function has the form `func(u, fval, user_data)`. (For full details see §4.6.1.)

`tmpl` (`N_Vector`) is any `N_Vector` (e.g. the initial guess vector `u`) which is used as a template to create (by cloning) necessary vectors in `kin_mem`.

Return value The return value `flag` (of type `int`) will be one of the following:

`KIN_SUCCESS`	The call to `KINInit` was successful.
`KIN_MEM_NULL`	The KINSOL memory block was not initialized through a previous call to `KINCreate`.
`KIN_MEM_FAIL`	A memory allocation request has failed.
`KIN_ILL_INPUT`	An input argument to `KINInit` has an illegal value.

Notes If an error occurred, `KINInit` sends an error message to the error handler function.

`KINFree`

Call `KINFree(&kin_mem);`

Description The function `KINFree` frees the memory allocated by a previous call to `KINCreate`.

Arguments The argument is the address of the pointer to the KINSOL memory block returned by `KINCreate` (of type `void *`).

Return value The function `KINFree` has no return value.

4.5.2 Linear solver specification functions

As previously explained, Newton and Picard iterations require the solution of linear systems of the form $J\delta = -F$. There are two KINSOL linear solvers currently available for this task: KINDLS and KINSPILS.

The first corresponds to the use of Direct Linear Solvers, and utilizes SUNMATRIX objects to store the Jacobian $J = \partial F/\partial u$ and factorizations used throughout the solution process.

The second corresponds to the use of Scaled, Preconditioned, Iterative Linear Solvers, utilizing matrix-free Krylov methods to solve the linear systems of equations. With most of these methods, preconditioning can be done on the left only, on the right only, on both the left and the right, or not at all. The exceptions to this rule are SPFGMR that supports right preconditioning only and PCG that performs symmetric preconditioning. For the specification of a preconditioner, see the iterative linear solver sections in §4.5.4 and §4.6. A preconditioner matrix P must approximate the Jacobian J, at least crudely.

With any of the Krylov solvers, only right preconditioning is supported through the KINSPILS interfaces. For specification of the preconditioner, see the Krylov solver sections within §4.5.4 and §4.6. If preconditioning is done, user-supplied functions define the right preconditioner matrix P, which should approximate the system Jacobian matrix J. For the specification of a preconditioner, see the iterative linear solver sections in §4.5.4 and §4.6. A preconditioner matrix P must approximate the Jacobian J, at least crudely.

To specify a generic linear solver to KINSOL, after the call to `KINCreate` but before any calls to `KINSol`, the user's program must create the appropriate SUNLINSOL object and call either of the functions `KINDlsSetLinearSolver` or `KINSpilsSetLinearSolver`, as documented below. The first argument passed to these functions is the KINSOL memory pointer returned by `KINCreate`; the second argument passed to these functions is the desired SUNLINSOL object to use for solving Newton or Picard

systems. A call to one of these functions initializes the appropriate KINSOL linear solver interface, linking this to the main KINSOL solver, and allows the user to specify parameters which are specific to a particular solver interface. The use of each of the generic linear solvers involves certain constants and possibly some macros, that are likely to be needed in the user code. These are available in the corresponding header file associated with the specific SUNMATRIX or SUNLINSOL module in question, as described in Chapters 7 and 8.

`KINDlsSetLinearSolver`

Call `flag = KINDlsSetLinearSolver(kin_mem, LS, J);`

Description The function `KINDlsSetLinearSolver` attaches a direct SUNLINSOL object `LS` and corresponding template Jacobian SUNMATRIX object `J` to KINSOL, initializing the KINDLS direct linear solver interface.

The user's main program must include the `kinsol_direct.h` header file.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `LS` (`SUNLinearSolver`) SUNLINSOL object to use for solving Newton linear systems.
- `J` (`SUNMatrix`) SUNMATRIX object for used as a template for the Jacobian (must have a type compatible with the linear solver object).

Return value The return value `flag` (of type `int`) is one of
- `KINDLS_SUCCESS` The KINDLS initialization was successful.
- `KINDLS_MEM_NULL` The `kin_mem` pointer is `NULL`.
- `KINDLS_ILL_INPUT` The KINDLS solver is not compatible with the current NVECTOR module.
- `KINDLS_MEM_FAIL` A memory allocation request failed.

Notes The KINDLS linear solver is not compatible with all implementations of the SUNLINSOL and NVECTOR modules. Specifically, KINDLS requires use of a *direct* SUNLINSOL object and a serial or theaded NVECTOR module. Additional compatibility limitations for each SUNLINSOL object (i.e. SUNMATRIX and NVECTOR object compatibility) are described in Chapter 8.

`KINSpilsSetLinearSolver`

Call `flag = KINSpilsSetLinearSolver(kin_mem, LS);`

Description The function `KINSpilsSetLinearSolver` attaches an iterative SUNLINSOL object `LS` to KINSOL, initializing the KINSPILS scaled, preconditioned, iterative linear solver interface.

The user's main program must include the `kin_spils.h` header file.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `LS` (`SUNLinearSolver`) SUNLINSOL object to use for solving Newton linear systems.

Return value The return value `flag` (of type `int`) is one of
- `KINSPILS_SUCCESS` The KINSPILS initialization was successful.
- `KINSPILS_MEM_NULL` The `kin_mem` pointer is `NULL`.
- `KINSPILS_ILL_INPUT` The KINSPILS solver is not compatible with the `LS` object or is incompatible with the current NVECTOR module.
- `KINSPILS_MEM_FAIL` A memory allocation request failed.
- `KINSPILS_SUNLS_FAIL` A call to the `LS` object failed.

Notes The KINSPILS linear solver interface is not compatible with all implementations of the SUNLINSOL and NVECTOR modules. Specifically, KINSPILS requires use of an *iterative* SUNLINSOL object. Additional compatibility limitations for each SUNLINSOL object (i.e. required NVECTOR routines) are described in Chapter 8.

4.5.3 KINSOL solver function

This is the central step in the solution process, the call to solve the nonlinear algebraic system.

`KINSol`

Call `flag = KINSol(kin_mem, u, strategy, u_scale, f_scale);`

Description The function `KINSol` computes an approximate solution to the nonlinear system.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `u` (`N_Vector`) vector set to initial guess by user before calling `KINSol`, but which upon return contains an approximate solution of the nonlinear system $F(u) = 0$.
- `strategy` (`int`) strategy used to solve the nonlinear system. It must be of the following:
  `KIN_NONE` basic Newton iteration
  `KIN_LINESEARCH` Newton with globalization
  `KIN_FP` fixed-point iteration with Anderson Acceleration (no linear solver needed)
  `KIN_PICARD` Picard iteration with Anderson Acceleration (uses a linear solver)
- `u_scale` (`N_Vector`) vector containing diagonal elements of scaling matrix D_u for vector `u` chosen so that the components of D_u·`u` (as a matrix multiplication) all have roughly the same magnitude when `u` is close to a root of $F(u)$.
- `f_scale` (`N_Vector`) vector containing diagonal elements of scaling matrix D_F for $F(u)$ chosen so that the components of $D_F \cdot F($`u`$)$ (as a matrix multiplication) all have roughly the same magnitude when `u` is not too near a root of $F(u)$. In the case of a fixed-point iteration, consider $F(u) = G(u) - u$.

Return value On return, `KINSol` returns the approximate solution in the vector `u` if successful. The return value `flag` (of type `int`) will be one of the following:

`KIN_SUCCESS`
: `KINSol` succeeded; the scaled norm of $F(u)$ is less than `fnormtol`.

`KIN_INITIAL_GUESS_OK`
: The guess `u` $= u_0$ satisfied the system $F(u) = 0$ within the tolerances specified.

`KIN_STEP_LT_STPTOL`
: KINSOL stopped based on scaled step length. This means that the current iterate may be an approximate solution of the given nonlinear system, but it is also quite possible that the algorithm is "stalled" (making insufficient progress) near an invalid solution, or that the scalar `scsteptol` is too large (see `KINSetScaledStepTol` in §4.5.4 to change `scsteptol` from its default value).

`KIN_MEM_NULL`
: The KINSOL memory block pointer was `NULL`.

`KIN_ILL_INPUT`
: An input parameter was invalid.

`KIN_NO_MALLOC`
: The KINSOL memory was not allocated by a call to `KINCreate`.

`KIN_MEM_FAIL`
: A memory allocation failed.

`KIN_LINESEARCH_NONCONV`
: The line search algorithm was unable to find an iterate sufficiently distinct from the current iterate, or could not find an iterate satisfying the sufficient decrease condition.

Failure to satisfy the sufficient decrease condition could mean the current iterate is "close" to an approximate solution of the given nonlinear system, the difference approximation of the matrix-vector product $J(u)v$ is inaccurate, or the real scalar `scsteptol` is too large.

`KIN_MAXITER_REACHED`

The maximum number of nonlinear iterations has been reached.

`KIN_MXNEWT_5X_EXCEEDED`

Five consecutive steps have been taken that satisfy the inequality $\|D_u p\|_{L2} > 0.99$ `mxnewtstep`, where p denotes the current step and `mxnewtstep` is a scalar upper bound on the scaled step length. Such a failure may mean that $\|D_F F(u)\|_{L2}$ asymptotes from above to a positive value, or the real scalar `mxnewtstep` is too small.

`KIN_LINESEARCH_BCFAIL`

The line search algorithm was unable to satisfy the "beta-condition" for `MXNBCF` +1 nonlinear iterations (not necessarily consecutive), which may indicate the algorithm is making poor progress.

`KIN_LINSOLV_NO_RECOVERY`

The user-supplied routine `psolve` encountered a recoverable error, but the preconditioner is already current.

`KIN_LINIT_FAIL`

The linear solver initialization routine (`linit`) encountered an error.

`KIN_LSETUP_FAIL`

The user-supplied routine `pset` (used to set up the preconditioner data) encountered an unrecoverable error.

`KIN_LSOLVE_FAIL`

Either the user-supplied routine `psolve` (used to to solve the preconditioned linear system) encountered an unrecoverable error, or the linear solver routine (`lsolve`) encountered an error condition.

`KIN_SYSFUNC_FAIL`

The system function failed in an unrecoverable manner.

`KIN_FIRST_SYSFUNC_ERR`

The system function failed recoverably at the first call.

`KIN_REPTD_SYSFUNC_ERR`

The system function had repeated recoverable errors. No recovery is possible.

Notes The components of vectors `u_scale` and `f_scale` should be strictly positive.

`KIN_SUCCESS` = 0, `KIN_INITIAL_GUESS_OK` = 1, and `KIN_STEP_LT_STPTOL` = 2. All remaining return values are negative and therefore a test `flag` < 0 will trap all `KINSol` failures.

4.5.4 Optional input functions

There are numerous optional input parameters that control the behavior of the KINSOL solver. KINSOL provides functions that can be used to change these from their default values. Table 4.2 lists all optional input functions in KINSOL which are then described in detail in the remainder of this section, beginning with those for the main KINSOL solver and continuing with those for the linear solver modules. For the most casual use of KINSOL, the reader can skip to §4.6.

We note that, on error return, all of these functions also send an error message to the error handler function. We also note that all error return values are negative, so a test `flag` < 0 will catch any error.

Table 4.2: Optional inputs for KINSOL, KINDLS, and KINSPILS

Optional input	**Function name**	**Default**
KINSOL main solver		
Error handler function	`KINSetErrHandlerFn`	internal fn.
Pointer to an error file	`KINSetErrFile`	`stderr`
Info handler function	`KINSetInfoHandlerFn`	internal fn.
Pointer to an info file	`KINSetInfoFile`	`stdout`
Data for problem-defining function	`KINSetUserData`	`NULL`
Verbosity level of output	`KINSetPrintLevel`	0
Max. number of nonlinear iterations	`KINSetNumMaxIters`	200
No initial matrix setup	`KINSetNoInitSetup`	`SUNFALSE`
No residual monitoring*	`KINSetNoResMon`	`SUNFALSE`
Max. iterations without matrix setup	`KINSetMaxSetupCalls`	10
Max. iterations without residual check*	`KINSetMaxSubSetupCalls`	5
Form of η coefficient	`KINSetEtaForm`	`KIN_ETACHOICE1`
Constant value of η	`KINSetEtaConstValue`	0.1
Values of γ and α	`KINSetEtaParams`	0.9 and 2.0
Values of ω_{min} and ω_{max}*	`KINSetResMonParams`	0.00001 and 0.9
Constant value of ω*	`KINSetResMonConstValue`	0.9
Lower bound on ϵ	`KINSetNoMinEps`	`SUNFALSE`
Max. scaled length of Newton step	`KINSetMaxNewtonStep`	$1000\|D_u u_0\|_2$
Max. number of β-condition failures	`KINSetMaxBetaFails`	10
Rel. error for D.Q. Jv	`KINSetRelErrFunc`	$\sqrt{\text{uround}}$
Function-norm stopping tolerance	`KINSetFuncNormTol`	$\text{uround}^{1/3}$
Scaled-step stopping tolerance	`KINSetScaledSteptol`	$\text{uround}^{2/3}$
Inequality constraints on solution	`KINSetConstraints`	`NULL`
Nonlinear system function	`KINSetSysFunc`	none
Anderson Acceleration subspace size	`KINSetMAA`	0
KINDLS linear solvers		
Jacobian function	`KINDlsSetJacFn`	DQ
KINSPILS linear solvers		
Preconditioner functions and data	`KINSpilsSetPreconditioner`	`NULL`, `NULL`, `NULL`
Jacobian-times-vector function and data	`KINSpilsSetJacTimesVecFn`	internal DQ, `NULL`

4.5.4.1 Main solver optional input functions

The calls listed here can be executed in any order. However, if either of the functions `KINSetErrFile` or `KINSetErrHandlerFn` is to be called, that call should be first, in order to take effect for any later error message.

`KINSetErrFile`

Call `flag = KINSetErrFile(kin_mem, errfp);`

Description The function `KINSetErrFile` specifies the pointer to the file where all KINSOL messages should be directed when the default KINSOL error handler function is used.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`errfp` (`FILE *`) pointer to output file.

Return value The return value `flag` (of type `int`) is one of

`KIN_SUCCESS` The optional value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

Notes The default value for `errfp` is `stderr`.

Passing a value of `NULL` disables all future error message output (except for the case in which the KINSOL memory pointer is `NULL`). This use of `KINSetErrFile` is strongly discouraged.

If `KINSetErrFile` is to be called, it should be called before any other optional input functions, in order to take effect for any later error message.

`KINSetErrHandlerFn`

Call `flag = KINSetErrHandlerFn(kin_mem, ehfun, eh_data);`

Description The function `KINSetErrHandlerFn` specifies the optional user-defined function to be used in handling error messages.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`ehfun` (`KINErrHandlerFn`) is the user's C error handler function (see §4.6.2).
`eh_data` (`void *`) pointer to user data passed to `ehfun` every time it is called.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The function `ehfun` and data pointer `eh_data` have been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

Notes The default internal error handler function directs error messages to the file specified by the file pointer `errfp` (see `KINSetErrFile` above).

Error messages indicating that the KINSOL solver memory is `NULL` will always be directed to `stderr`.

`KINSetInfoFile`

Call `flag = KINSetInfoFile(kin_mem, infofp);`

Description The function `KINSetInfoFile` specifies the pointer to the file where all informative (non-error) messages should be directed.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`infofp` (`FILE *`) pointer to output file.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The optional value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

Notes The default value for `infofp` is `stdout`.

`KINSetInfoHandlerFn`

Call `flag = KINSetInfoHandlerFn(kin_mem, ihfun, ih_data);`

Description The function `KINSetInfoHandlerFn` specifies the optional user-defined function to be used in handling informative (non-error) messages.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`ihfun` (`KINInfoHandlerFn`) is the user's C information handler function (see §4.6.3).
`ih_data` (`void *`) pointer to user data passed to `ihfun` every time it is called.

Return value The return value `flag` (of type `int`) is one of:
`KIN_SUCCESS` The function `ihfun` and data pointer `ih_data` have been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

Notes The default internal information handler function directs informative (non-error) messages to the file specified by the file pointer `infofp` (see `KINSetInfoFile` above).

`KINSetPrintLevel`

Call `flag = KINSetPrintLevel(kin_mem, printfl);`

Description The function `KINSetPrintLevel` specifies the level of verbosity of the output.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`printfl` (`int`) flag indicating the level of verbosity. Must be one of:

- 0 no information displayed.
- 1 for each nonlinear iteration display the following information: the scaled Euclidean ℓ_2 norm of the system function evaluated at the current iterate, the scaled norm of the Newton step (only if using `KIN_NONE`), and the number of function evaluations performed so far.
- 2 display level 1 output and the following values for each iteration:
  $\|F(u)\|_{D_F}$ (only for `KIN_NONE`).
  $\|F(u)\|_{D_F,\infty}$ (for `KIN_NONE` and `KIN_LINESEARCH`).
- 3 display level 2 output plus additional values used by the global strategy (only if using `KIN_LINESEARCH`), and statistical information for the linear solver.

Return value The return value `flag` (of type `int`) is one of:
`KIN_SUCCESS` The optional value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT` The argument `printfl` had an illegal value.

Notes The default value for `printfl` is 0.

`KINSetUserData`

Call `flag = KINSetUserData(kin_mem, user_data);`

Description The function `KINSetUserData` specifies the pointer to user-defined memory that is to be passed to all user-supplied functions.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`user_data` (`void *`) pointer to the user-defined memory.

Return value The return value `flag` (of type `int`) is one of:
`KIN_SUCCESS` The optional value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

Notes If specified, the pointer to `user_data` is passed to all user-supplied functions that have it as an argument. Otherwise, a `NULL` pointer is passed.

If `user_data` is needed in user linear solver or preconditioner functions, the call to `KINSetUserData` must be made *before* the call to specify the linear solver.

`KINSetNumMaxIters`

Call `flag = KINSetNumMaxIters(kin_mem, mxiter);`

Description The function `KINSetNumMaxIters` specifies the maximum number of nonlinear iterations allowed.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`mxiter` (`long int`) maximum number of nonlinear iterations.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The optional value has been successfully set.

`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

`KIN_ILL_INPUT` The maximum number of iterations was non-positive.

Notes The default value for `mxiter` is `MXITER_DEFAULT` = 200.

`KINSetNoInitSetup`

Call `flag = KINSetNoInitSetup(kin_mem, noInitSetup);`

Description The function `KINSetNoInitSetup` specifies whether an initial call to the preconditioner or Jacobian setup function should be made or not.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`noInitSetup` (`booleantype`) flag controlling whether an initial call to the preconditioner or Jacobian setup function is made (pass `SUNFALSE`) or not made (pass `SUNTRUE`).

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The optional value has been successfully set.

`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

Notes The default value for `noInitSetup` is `SUNFALSE`, meaning that an initial call to the preconditioner or Jacobian setup function will be made.

A call to this function is useful when solving a sequence of problems, in which the final preconditioner or Jacobian value from one problem is to be used initially for the next problem.

`KINSetNoResMon`

Call `flag = KINSetNoResMon(kin_mem, noNNIResMon);`

Description The function `KINSetNoResMon` specifies whether or not the nonlinear residual monitoring scheme is used to control Jacobian updating

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`noNNIResMon` (`booleantype`) flag controlling whether residual monitoring is used (pass `SUNFALSE`) or not used (pass `SUNTRUE`).

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The optional value has been successfully set.

`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

Notes When using a direct solver, the default value for `noNNIResMon` is `SUNFALSE`, meaning that the nonlinear residual will be monitored.

Residual monitoring is only available for use with the direct linear solver modules (meaning KINDENSE, KINBAND, KINKLU, and KINSUPERLUMT).

`KINSetMaxSetupCalls`

Call `flag = KINSetMaxSetupCalls(kin_mem, msbset);`

Description The function `KINSetMaxSetupCalls` specifies the maximum number of nonlinear iterations that can be performed between calls to the preconditioner or Jacobian setup function.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`msbset` (`long int`) maximum number of nonlinear iterations without a call to the preconditioner or Jacobian setup function. Pass 0 to indicate the default.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT`	The argument `msbset` was negative.

Notes The default value for `msbset` is `MSBSET_DEFAULT` $= 10$.

`KINSetMaxSubSetupCalls`

Call `flag = KINSetMaxSubSetupCalls(kin_mem, msbsetsub);`

Description The function `KINSetMaxSubSetupCalls` specifies the maximum number of nonlinear iterations between checks by the residual monitoring algorithm.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`msbsetsub` (`long int`) maximum number of nonlinear iterations without checking the nonlinear residual. Pass 0 to indicate the default.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT`	The argument `msbsetsub` was negative.

Notes The default value for `msbsetsub` is `MSBSET_SUB_DEFAULT` $= 5$.

Residual monitoring is only available for use with the direct linear solver modules (meaning KINDENSE, KINBAND, KINKLU, and KINSUPERLUMT).

`KINSetEtaForm`

Call `flag = KINSetEtaForm(kin_mem, etachoice);`

Description The function `KINSetEtaForm` specifies the method for computing the value of the η coefficient used in the calculation of the linear solver convergence tolerance.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`etachoice` (`int`) flag indicating the method for computing η. The value must be one of `KIN_ETACHOICE1`, `KIN_ETACHOICE2`, or `KIN_ETACONSTANT` (see Chapter 2 for details).

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.

KIN_ILL_INPUT The argument etachoice had an illegal value.

Notes The default value for etachoice is KIN_ETACHOICE1.

KINSetEtaConstValue

Call `flag = KINSetEtaConstValue(kin_mem, eta);`

Description The function KINSetEtaConstValue specifies the constant value for η in the case etachoice = KIN_ETACONSTANT.

Arguments kin_mem (void *) pointer to the KINSOL memory block.
eta (realtype) constant value for η. Pass 0.0 to indicate the default.

Return value The return value flag (of type int) is one of:

KIN_SUCCESS The optional value has been successfully set.
KIN_MEM_NULL The kin_mem pointer is NULL.
KIN_ILL_INPUT The argument eta had an illegal value

Notes The default value for eta is 0.1. The legal values are $0.0 <$ eta ≤ 1.0.

KINSetEtaParams

Call `flag = KINSetEtaParams(kin_mem, egamma, ealpha);`

Description The function KINSetEtaParams specifies the parameters γ and α in the formula for η, in the case etachoice = KIN_ETACHOICE2.

Arguments kin_mem (void *) pointer to the KINSOL memory block.
egamma (realtype) value of the γ parameter. Pass 0.0 to indicate the default.
ealpha (realtype) value of the α parameter. Pass 0.0 to indicate the default.

Return value The return value flag (of type int) is one of:

KIN_SUCCESS The optional values have been successfully set.
KIN_MEM_NULL The kin_mem pointer is NULL.
KIN_ILL_INPUT One of the arguments egamma or ealpha had an illegal value.

Notes The default values for egamma and ealpha are 0.9 and 2.0, respectively.

The legal values are $0.0 <$ egamma ≤ 1.0 and $1.0 <$ ealpha ≤ 2.0.

KINSetResMonConstValue

Call `flag = KINSetResMonConstValue(kin_mem, omegaconst);`

Description The function KINSetResMonConstValue specifies the constant value for ω when using residual monitoring.

Arguments kin_mem (void *) pointer to the KINSOL memory block.
omegaconst (realtype) constant value for ω. Passing 0.0 results in using Eqn. (2.4).

Return value The return value flag (of type int) is one of:

KIN_SUCCESS The optional value has been successfully set.
KIN_MEM_NULL The kin_mem pointer is NULL.
KIN_ILL_INPUT The argument omegaconst had an illegal value

Notes The default value for omegaconst is 0.9. The legal values are $0.0 <$ omegaconst < 1.0.

`KINSetResMonParams`

Call `flag = KINSetResMonParams(kin_mem, omegamin, omegamax);`

Description The function `KINSetResMonParams` specifies the parameters ω_{min} and ω_{max} in the formula (2.4) for ω.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`omegamin` (`realtype`) value of the ω_{min} parameter. Pass 0.0 to indicate the default.
`omegamax` (`realtype`) value of the ω_{max} parameter. Pass 0.0 to indicate the default.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional values have been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT`	One of the arguments `omegamin` or `omegamax` had an illegal value.

Notes The default values for `omegamin` and `omegamax` are 0.00001 and 0.9, respectively.
The legal values are $0.0 <$ `omegamin` $<$ `omegamax` < 1.0.

`KINSetNoMinEps`

Call `flag = KINSetNoMinEps(kin_mem, noMinEps);`

Description The function `KINSetNoMinEps` specifies a flag that controls whether or not the value of ϵ, the scaled linear residual tolerance, is bounded from below.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`noMinEps` (`booleantype`) flag controlling the bound on ϵ.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.

Notes The default value for `noMinEps` is `SUNFALSE`, meaning that a positive minimum value, equal to 0.01*`fnormtol`, is applied to ϵ. (See `KINSetFuncNormTol` below.)

`KINSetMaxNewtonStep`

Call `flag = KINSetMaxNewtonStep(kin_mem, mxnewtstep);`

Description The function `KINSetMaxNewtonStep` specifies the maximum allowable scaled length of the Newton step.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`mxnewtstep` (`realtype`) maximum scaled step length (≥ 0.0). Pass 0.0 to indicate the default.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT`	The input value was negative.

Notes The default value of `mxnewtstep` is $1000 \|u_0\|_{D_u}$, where u_0 is the initial guess.

`KINSetMaxBetaFails`

Call `flag = KINSetMaxBetaFails(kin_mem, mxnbcf);`

Description The function `KINSetMaxBetaFails` specifies the maximum number of β-condition failures in the linesearch algorithm.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`mxnbcf` (`realtype`) maximum number of β-condition failures. Pass 0.0 to indicate the default.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The optional value has been successfully set.

`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

`KIN_ILL_INPUT` `mxnbcf` was negative.

Notes The default value of `mxnbcf` is `MXNBCF_DEFAULT` $= 10$.

`KINSetRelErrFunc`

Call `flag = KINSetRelErrFunc(kin_mem, relfunc);`

Description The function `KINSetRelErrFunc` specifies the relative error in computing $F(u)$, which is used in the difference quotient approximation to the Jacobian matrix [see Eq.(2.7)] or the Jacobian-vector product [see Eq.(2.9)].

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`relfunc` (`realtype`) relative error in $F(u)$ (`relfunc` ≥ 0.0). Pass 0.0 to indicate the default.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The optional value has been successfully set.

`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

`KIN_ILL_INPUT` The relative error was negative.

Notes The default value for `relfunc` is U = unit roundoff.

`KINSetFuncNormTol`

Call `flag = KINSetFuncNormTol(kin_mem, fnormtol);`

Description The function `KINSetFuncNormTol` specifies the scalar used as a stopping tolerance on the scaled maximum norm of the system function $F(u)$.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`fnormtol` (`realtype`) tolerance for stopping based on scaled function norm (≥ 0.0). Pass 0.0 to indicate the default.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The optional value has been successfully set.

`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

`KIN_ILL_INPUT` The tolerance was negative.

Notes The default value for `fnormtol` is (unit roundoff)$^{1/3}$.

`KINSetScaledStepTol`

Call `flag = KINSetScaledStepTol(kin_mem, scsteptol);`

Description The function `KINSetScaledStepTol` specifies the scalar used as a stopping tolerance on the minimum scaled step length.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`scsteptol` (`realtype`) tolerance for stopping based on scaled step length (≥ 0.0). Pass 0.0 to indicate the default.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS` The optional value has been successfully set.

KIN_MEM_NULL The kin_mem pointer is NULL.

KIN_ILL_INPUT The tolerance was non-positive.

Notes The default value for scsteptol is (unit roundoff)$^{2/3}$.

KINSetConstraints

Call flag = KINSetConstraints(kin_mem, constraints);

Description The function KINSetConstraints specifies a vector that defines inequality constraints for each component of the solution vector u.

Arguments kin_mem (void *) pointer to the KINSOL memory block.

constraints (N_Vector) vector of constraint flags. If constraints[i] is

- 0.0 then no constraint is imposed on u_i.
- 1.0 then u_i will be constrained to be $u_i \geq 0.0$.
- −1.0 then u_i will be constrained to be $u_i \leq 0.0$.
- 2.0 then u_i will be constrained to be $u_i > 0.0$.
- −2.0 then u_i will be constrained to be $u_i < 0.0$.

Return value The return value flag (of type int) is one of:

KIN_SUCCESS The optional value has been successfully set.

KIN_MEM_NULL The kin_mem pointer is NULL.

KIN_ILL_INPUT The constraint vector contains illegal values.

Notes The presence of a non-NULL constraints vector that is not 0.0 in all components will cause constraint checking to be performed.

The function creates a private copy of the constraints vector. Consequently, the user-supplied vector can be freed after the function call, and the constraints can only be changed by calling this function.

KINSetSysFunc

Call flag = KINSetSysFunc(kin_mem, func);

Description The function KINSetSysFunc specifies the user-provided function that evaluates the nonlinear system function $F(u)$ or $G(u)$.

Arguments kin_mem (void *) pointer to the KINSOL memory block.

func (KINSysFn) user-supplied function that evaluates $F(u)$ (or $G(u)$ for fixed-point iteration).

Return value The return value flag (of type int) is one of:

KIN_SUCCESS The optional value has been successfully set.

KIN_MEM_NULL The kin_mem pointer is NULL.

KIN_ILL_INPUT The argument func was NULL.

Notes The nonlinear system function is initially specified through KINInit. The option of changing the system function is provided for a user who wishes to solve several problems of the same size but with different functions.

KINSetMAA

Call flag = KINSetMAA(kin_mem, maa);

Description The function KINSetMAA specifies the size of the subspace used with Anderson acceleration in conjunction with Picard or fixed-point iteration.

Arguments kin_mem (void *) pointer to the KINSOL memory block.

maa (long int) subspace size for various methods. A value of 0 means no acceleration, while a positive value means acceleration will be done.

Return value The return value flag (of type int) is one of:

KIN_SUCCESS The optional value has been successfully set.

KIN_MEM_NULL The kin_mem pointer is NULL.

KIN_ILL_INPUT The argument maa was negative.

Notes This function sets the subspace size, which needs to be > 0 if Anderson Acceleration is to be used. It also allocates additional memory necessary for Anderson Acceleration.

The default value of maa is 0, indicating no acceleration. The value of maa should always be less than mxiter.

This function MUST be called before calling KINInit.

If the user calls the function KINSetNumMaxIters, that call should be made before the call to KINSetMAA, as the latter uses the value of mxiter.

4.5.4.2 Direct linear solver interface optional input functions

The KINDLS solver interface needs a function to compute an approximation to the Jacobian matrix $J(u)$. This function must be of type KINDlsJacFn. The user can supply a Jacobian function, or if using a dense or banded matrix J can use the default internal difference quotient approximation that comes with the KINDLS solver. To specify a user-supplied Jacobian function jac, KINDLS provides the function KINDlsSetJacFn. The KINDLS interface passes the pointer user_data to the Jacobian function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The pointer user_data may be specified through KINSetUserData.

KINDlsSetJacFn

Call flag = KINDlsSetJacFn(ida_mem, jac);

Description The function KINDlsSetJacFn specifies the Jacobian approximation function to be used.

Arguments kin_mem (void *) pointer to the KINSOL memory block.

jac (KINDlsJacFn) user-defined Jacobian approximation function.

Return value The return value flag (of type int) is one of

KINDLS_SUCCESS The optional value has been successfully set.

KINDLS_MEM_NULL The kin_mem pointer is NULL.

KINDLS_LMEM_NULL The KINDLS linear solver interface has not been initialized.

Notes By default, KINDLS uses an internal difference quotient function for dense and band matrices. If NULL is passed to jac, this default function is used. An error will occur if no jac is supplied when using a sparse matrix.

The function type KINDlsJacFn is described in §4.6.4.

4.5.4.3 Iterative linear solver interface optional input functions

If any preconditioning is to be done with one of the KINSPILS linear solver interfaces, then the user must supply a preconditioner solve function, psolve, and specify its name in a call to to the routine KINSpilsSetPreconditioner. The evaluation and preprocessing of any Jacobian-related data needed by the user's preconditioner solve function is done in the optional user-supplied function psetup. Both of these functions are fully specified in §4.6. If used, the psetup function should also be specified in the call to KINSpilsSetPreconditioner. A KINSPILS solver passes the pointer user_data received through KINSetUserData to the preconditioner psetup and psolve functions. This allows the user

to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied preconditioner functions without using global data in the program.

Ther KINSPILS solvers require a function to compute an approximation to the product between the Jacobian matrix $J(u)$ and a vector v. The user can supply his/her own Jacobian-times-vector approximation function, or use the internal difference quotient approximation that comes with the KINSPILS solver interface. A user-defined Jacobian-vector function must be of type `KINSpilsJacTimesVecFn` and can be specified through a call to `KINSpilsSetJacTimes` (see §4.6.5 for specification details). A KINSPILS solver passes the pointer `user_data` received through `KINSetUserData` to the Jacobian-times-vector function `jtimes` each time it is called.

`KINSpilsSetPreconditioner`

Call `flag = KINSpilsSetPreconditioner(kin_mem, psetup, psolve);`

Description The function `KINSpilsSetPreconditioner` specifies the preconditioner setup and solve functions.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`psetup` (`KINSpilsPrecSetupFn`) user-defined function to set up the preconditioner. Pass `NULL` if no setup operation is necessary.

`psolve` (`KINSpilsPrecSolveFn`) user-defined preconditioner solve function.

Return value The return value `flag` (of type `int`) is one of

`KINSPILS_SUCCESS`	The optional values have been successfully set.
`KINSPILS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSPILS_LMEM_NULL`	The KINSPILS linear solver has not been initialized.
`KINSPILS_SUNLS_FAIL`	An error occurred when setting up preconditioning in the SUNLINSOL object used by the KINSPILS interface.

Notes The function type `KINSpilsPrecSolveFn` is described in §4.6.6. The function type `KINSpilsPrecSetupFn` is described in §4.6.7.

`KINSpilsSetJacTimes`

Call `flag = KINSpilsSetJacTimes(kin_mem, jsetup, jtimes);`

Description The function `KINSpilsSetJacTimes` specifies the Jacobian-vector setup and product.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`jtsetup` (`KINSpilsJacTimesSetupFn`) user-defined function to set up the Jacobian-vector product. Pass `NULL` if no setup is necessary.

`jtimes` (`KINSpilsJacTimesVecFn`) user-defined Jacobian-vector product function.

Return value The return value `flag` (of type `int`) is one of

`KINSPILS_SUCCESS`	The optional value has been successfully set.
`KINSPILS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSPILS_LMEM_NULL`	The KINSPILS linear solver has not been initialized.
`KINSPILS_SUNLS_FAIL`	An error occurred when setting up the system matrix-times-vector routines in the SUNLINSOL object used by the KINSPILS interface.

Notes By default, the KINSPILS linear solvers use an internal difference quotient function `KINSpilsDQJtimes`. If `NULL` is passed as `jtimes`, this default function is used.

The function type `KINSpilsJacTimesVecFn` is described in §4.6.5.

Table 4.3: Optional outputs from KINSOL, KINDLS, and KINSPILS

Optional output	Function name
KINSOL main solver	
Size of KINSOL real and integer workspaces	KINGetWorkSpace
Number of function evaluations	KINGetNumFuncEvals
Number of nonlinear iterations	KINGetNumNolinSolvIters
Number of β-condition failures	KINGetNumBetaCondFails
Number of backtrack operations	KINGetNumBacktrackOps
Scaled norm of F	KINGetFuncNorm
Scaled norm of the step	KINGetStepLength
KINDLS linear solver interface	
Size of real and integer workspaces	KINDlsGetWorkSpace
No. of Jacobian evaluations	KINDlsGetNumJacEvals
No. of F calls for D.Q. Jacobian evals.	KINDlsGetNumFuncEvals
Last return from a KINDLS function	KINDlsGetLastFlag
KINSPILS linear solver interface	
Size of real and integer workspaces	KINSpilsGetWorkSpace
No. of linear iterations	KINSpilsGetNumLinIters
No. of linear convergence failures	KINSpilsGetNumConvFails
No. of preconditioner evaluations	KINSpilsGetNumPrecEvals
No. of preconditioner solves	KINSpilsGetNumPrecSolves
No. of Jacobian-vector setup evaluations	IDASpilsGetNumJTSetupEvals
No. of Jacobian-vector product evaluations	KINSpilsGetNumJtimesEvals
No. of F calls for D.Q. Jacobian-vector evals.	KINSpilsGetNumFuncEvals
Last return from a linear solver function	KINSpilsGetLastFlag

4.5.5 Optional output functions

KINSOL provides an extensive list of functions that can be used to obtain solver performance information. Table 4.3 lists all optional output functions in KINSOL, which are then described in detail in the remainder of this section, beginning with those for the main KINSOL solver and continuing with those for the linear solver modules. Where the name of an output from a linear solver module would otherwise conflict with the name of an optional output from the main solver, a suffix `LS` (for Linear Solver) has been added here (*e.g.*, `lenrwLS`).

4.5.5.1 SUNDIALS version information

The following functions provide a way to get SUNDIALS version information at runtime.

`SUNDIALSGetVersion`

Call `flag = SUNDIALSGetVersion(version, len);`

Description The function `SUNDIALSGetVersion` fills a character array with SUNDIALS version information.

Arguments `version` (`char *`) character array to hold the SUNDIALS version information.
`len` (`int`) allocated length of the `version` character array.

Return value If successful, `SUNDIALSGetVersion` returns 0 and `version` contains the SUNDIALS version information. Otherwise, it returns -1 and `version` is not set (the input character array is too short).

Notes A string of 25 characters should be sufficient to hold the version information. Any trailing characters in the `version` array are removed.

`SUNDIALSGetVersionNumber`

Call `flag = SUNDIALSGetVersionNumber(&major, &minor, &patch, label, len);`

Description The function `SUNDIALSGetVersionNumber` set integers for the SUNDIALS major, minor, and patch release numbers and fills a character array with the release label if applicable.

Arguments
- `major` (`int`) SUNDIALS release major version number.
- `minor` (`int`) SUNDIALS release minor version number.
- `patch` (`int`) SUNDIALS release patch version number.
- `label` (`char *`) character array to hold the SUNDIALS release label.
- `len` (`int`) allocated length of the `label` character array.

Return value If successful, `SUNDIALSGetVersionNumber` returns 0 and the `major`, `minor`, `patch`, and `label` values are set. Otherwise, it returns -1 and the values are not set (the input character array is too short).

Notes A string of 10 characters should be sufficient to hold the label information. If a label is not used in the release version, no information is copied to `label`. Any trailing characters in the `label` array are removed.

4.5.5.2 Main solver optional output functions

KINSOL provides several user-callable functions that can be used to obtain different quantities that may be of interest to the user, such as solver workspace requirements and solver performance statistics. These optional output functions are described next.

`KINGetWorkSpace`

Call `flag = KINGetWorkSpace(kin_mem, &lenrw, &leniw);`

Description The function `KINGetWorkSpace` returns the KINSOL integer and real workspace sizes.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `lenrw` (`long int`) the number of `realtype` values in the KINSOL workspace.
- `leniw` (`long int`) the number of integer values in the KINSOL workspace.

Return value The return value `flag` (of type `int`) is one of:
- `KIN_SUCCESS` The optional output values have been successfully set.
- `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

Notes In terms of the problem size N, the actual size of the real workspace is $17+5N$ `realtype` words. The real workspace is increased by an additional N words if constraint checking is enabled (see `KINSetConstraints`).

The actual size of the integer workspace (without distinction between `int` and `long int`) is $22 + 5N$ (increased by N if constraint checking is enabled).

`KINGetNumFuncEvals`

Call `flag = KINGetNumFuncEvals(kin_mem, &nfevals);`

Description The function `KINGetNumFuncEvals` returns the number of evaluations of the system function.

Arguments
- `kin_mem` (`void *`) pointer to the KINSOL memory block.
- `nfevals` (`long int`) number of calls to the user-supplied function that evaluates $F(u)$.

Return value The return value `flag` (of type `int`) is one of:
- `KIN_SUCCESS` The optional output value has been successfully set.
- `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

`KINGetNumNonlinSolvIters`

Call `flag = KINGetNumNonlinSolvIters(kin_mem, &nniters);`

Description The function `KINGetNumNonlinSolvIters` returns the number of nonlinear iterations.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`nniters` (`long int`) number of nonlinear iterations.

Return value The return value `flag` (of type `int`) is one of:
`KIN_SUCCESS` The optional output value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

`KINGetNumBetaCondFails`

Call `flag = KINGetNumBetaCondFails(kin_mem, &nbcfails);`

Description The function `KINGetNumBetaCondFails` returns the number of β-condition failures.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`nbcfails` (`long int`) number of β-condition failures.

Return value The return value `flag` (of type `int`) is one of:
`KIN_SUCCESS` The optional output value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

`KINGetNumBacktrackOps`

Call `flag = KINGetNumBacktrackOps(kin_mem, &nbacktr);`

Description The function `KINGetNumBacktrackOps` returns the number of backtrack operations (step length adjustments) performed by the line search algorithm.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`nbacktr` (`long int`) number of backtrack operations.

Return value The return value `flag` (of type `int`) is one of:
`KIN_SUCCESS` The optional output value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

`KINGetFuncNorm`

Call `flag = KINGetFuncNorm(kin_mem, &fnorm);`

Description The function `KINGetFuncNorm` returns the scaled Euclidean ℓ_2 norm of the nonlinear system function $F(u)$ evaluated at the current iterate.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`fnorm` (`realtype`) current scaled norm of $F(u)$.

Return value The return value `flag` (of type `int`) is one of:
`KIN_SUCCESS` The optional output value has been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

`KINGetStepLength`

Call `flag = KINGetStepLength(kin_mem, &steplength);`

Description The function `KINGetStepLength` returns the scaled Euclidean ℓ_2 norm of the step used during the previous iteration.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

steplength (realtype) scaled norm of the Newton step.

Return value The return value flag (of type int) is one of:

KIN_SUCCESS The optional output value has been successfully set.

KIN_MEM_NULL The kin_mem pointer is NULL.

4.5.5.3 Direct linear solver interface optional output functions

The following optional outputs are available from the KINDLS module: workspace requirements, number of calls to the Jacobian routine, number of calls to the system function routine for difference quotient Jacobian approximation, and last return value from a KINDLS function.

KINDlsGetWorkSpace

Call `flag = KINDlsGetWorkSpace(kin_mem, &lenrwLS, &leniwLS);`

Description The function KINDlsGetWorkSpace returns the KINDLS real and integer workspace sizes.

Arguments kin_mem (void *) pointer to the KINSOL memory block.

lenrwLS (long int) the number of realtype values in the KINDLS workspace.

leniwLS (long int) the number of integer values in the KINDLS workspace.

Return value The return value flag (of type int) is one of

KINDLS_SUCCESS The optional output value has been successfully set.

KINDLS_MEM_NULL The kin_mem pointer is NULL.

KINDLS_LMEM_NULL The KINDENSE linear solver has not been initialized.

Notes The workspace requirements reported by this routine correspond only to memory allocated within this interface and to memory allocated by the SUNLINSOL object attached to it. The template Jacobian matrix allocated by the user outside of KINDLS is not included in this report.

KINDlsGetNumJacEvals

Call `flag = KINDlsGetNumJacEvals(kin_mem, &njevals);`

Description The function KINDlsGetNumJacEvals returns the cummulative number of calls to the Jacobian approximation function.

Arguments kin_mem (void *) pointer to the KINSOL memory block.

njevals (long int) the number of calls to the Jacobian function.

Return value The return value flag (of type int) is one of

KINDLS_SUCCESS The optional output value has been successfully set.

KINDLS_MEM_NULL The kin_mem pointer is NULL.

KINDLS_LMEM_NULL The KINDENSE linear solver has not been initialized.

KINDlsGetNumFuncEvals

Call `flag = KINDlsGetNumFuncEvals(kin_mem, &nfevalsLS);`

Description The function KINDlsGetNumFuncEvals returns the number of calls to the user system function used to compute the difference quotient approximation to the Jacobian.

Arguments kin_mem (void *) pointer to the KINSOL memory block.

nfevalsLS (long int) the number of calls to the user system function.

Return value The return value flag (of type int) is one of

KINDLS_SUCCESS The optional output value has been successfully set.

KINDLS_MEM_NULL	The kin_mem pointer is NULL.
KINDLS_LMEM_NULL	The KINDENSE or KINBAND linear solver has not been initialized.

Notes The value nfevalsLS is incremented only if one of the default internal difference quotient functions (dense or banded) is used.

KINDlsGetLastFlag

Call `flag = KINDlsGetLastFlag(kin_mem, &lsflag);`

Description The function KINDlsGetLastFlag returns the last return value from a KINDENSE routine.

Arguments kin_mem (void *) pointer to the KINSOL memory block.

lsflag (long int) the value of the last return flag from a KINDENSE function.

Return value The return value flag (of type int) is one of

KINDLS_SUCCESS	The optional output value has been successfully set.
KINDLS_MEM_NULL	The kin_mem pointer is NULL.
KINDLS_LMEM_NULL	The KINDENSE linear solver has not been initialized.

Notes If the SUNLINSOL_DENSE or SUNLINSOL_BAND setup function failed (KINSolve returned KIN_LSETUP_FAIL), then the value of lsflag is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the (dense or banded) Jacobian matrix.

4.5.5.4 Iterative linear solver interface optional output functions

The following optional outputs are available from the KINSPILS modules: workspace requirements, number of linear iterations, number of linear convergence failures, number of calls to the preconditioner setup and solve routines, number of calls to the Jacobian-vector setup and product routines, number of calls to the system function routine for difference quotient Jacobian-vector product approximation, and last return value from a linear solver function.

KINSpilsGetWorkSpace

Call `flag = KINSpilsGetWorkSpace(kin_mem, &lenrwLS, &leniwLS);`

Description The function KINSpilsGetWorkSpace returns the global sizes of the linear solver real and integer workspaces.

Arguments kin_mem (void *) pointer to the KINSOL memory block.

lenrwLS (long int) the number of realtype values in the linear solver workspace.

leniwLS (long int) the number of integer values in the linear solver workspace.

Return value The return value flag (of type int) is one of:

KINSPILS_SUCCESS	The optional output values have been successfully set.
KINSPILS_MEM_NULL	The kin_mem pointer is NULL.
KINSPILS_LMEM_NULL	The linear solver has not been initialized.

Notes The workspace requirements reported by this routine correspond only to memory allocated within this interface and to memory allocated by the SUNLINSOL object attached to it.

In a parallel setting, the above values are global (i.e., summed over all processors).

`KINSpilsGetNumLinIters`

Call `flag = KINSpilsGetNumLinIters(kin_mem, &nliters);`

Description The function `KINSpilsGetNumLinIters` returns the cumulative number of linear iterations.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`nliters` (`long int`) the current number of linear iterations.

Return value The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS`	The optional output value has been successfully set.
`KINSPILS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSPLIS_LMEM_NULL`	The KINSPILS linear solver has not been initialized.

`KINSpilsGetNumConvFails`

Call `flag = KINSpilsGetNumConvFails(kin_mem, &nlcfails);`

Description The function `KINSpilsGetNumConvFails` returns the cumulative number of linear convergence failures.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`nlcfails` (`long int`) the current number of linear convergence failures.

Return value The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS`	The optional output value has been successfully set.
`KINSPILS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSPILS_LMEM_NULL`	The KINSPILS linear solver has not been initialized.

`KINSpilsGetNumPrecEvals`

Call `flag = KINSpilsGetNumPrecEvals(kin_mem, &npevals);`

Description The function `KINSpilsGetNumPrecEvals` returns the cumulative number of preconditioner evaluations, i.e., the number of calls made to `psetup`.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`npevals` (`long int`) the current number of calls to `psetup`.

Return value The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS`	The optional output value has been successfully set.
`KINSPILS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSPILS_LMEM_NULL`	The KINSPILS linear solver has not been initialized.

`KINSpilsGetNumPrecSolves`

Call `flag = KINSpilsGetNumPrecSolves(kin_mem, &npsolves);`

Description The function `KINSpilsGetNumPrecSolves` returns the cumulative number of calls made to the preconditioner solve function, `psolve`.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`npsolves` (`long int`) the current number of calls to `psolve`.

Return value The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS`	The optional output value has been successfully set.
`KINSPILS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSPILS_LMEM_NULL`	The linear solver module has not been initialized.

`KINSpilsGetNumJTSetupEvals`

Call `flag = KINSpilsGetNumJTSetupEvals(kin_mem, &njtsetup);`

Description The function `KINSpilsGetNumJTSetupEvals` returns the cumulative number of calls made to the Jacobian-vector setup function `jtsetup`.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`njtsetup` (`long int`) the current number of calls to `jtsetup`.

Return value The return value `flag` (of type `int`) is one of

`KINSPILS_SUCCESS` The optional output value has been successfully set.
`KINSPILS_MEM_NULL` The `ida_mem` pointer is `NULL`.
`KINSPILS_LMEM_NULL` The KINSPILS linear solver has not been initialized.

`KINSpilsGetNumJtimesEvals`

Call `flag = KINSpilsGetNumJtimesEvals(kin_mem, &njvevals);`

Description The function `KINSpilsGetNumJtimesEvals` returns the cumulative number made to the Jacobian-vector product function, `jtimes`.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`njvevals` (`long int`) the current number of calls to `jtimes`.

Return value The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS` The optional output value has been successfully set.
`KINSPILS_MEM_NULL` The `kin_mem` pointer is `NULL`.
`KINSPILS_LMEM_NULL` The KINSPILS linear solver has not been initialized.

`KINSpilsGetNumFuncEvals`

Call `flag = KINSpilsGetNumFuncEvals(kin_mem, &nfevalsLS);`

Description The function `KINSpilsGetNumFuncEvals` returns the number of calls to the user system function for difference quotient Jacobian-vector product approximations.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`nfevalsLS` (`long int`) the number of calls to the user system function.

Return value The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS` The optional output value has been successfully set.
`KINSPILS_MEM_NULL` The `kin_mem` pointer is `NULL`.
`KINSPILS_LMEM_NULL` The KINSPILS linear solver has not been initialized.

Notes The value `nfevalsLS` is incremented only if the default `KINSpilsDQJtimes` difference quotient function is used.

`KINSpilsGetLastFlag`

Call `flag = KINSpilsGetLastFlag(kin_mem, &lsflag);`

Description The function `KINSpilsGetLastFlag` returns the last return value from a KINSPILS routine.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`lsflag` (`long int`) the value of the last return flag from a KINSPILS function.

Return value The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS` The optional output value has been successfully set.
`KINSPILS_MEM_NULL` The `kin_mem` pointer is `NULL`.

KINSPILS_LMEM_NULL The KINSPILS linear solver has not been initialized.

Notes If the KINSPILS setup function failed (KINSOL returned KIN_LSETUP_FAIL), lsflag will be set to SUNLS_PSET_FAIL_UNREC, SUNLS_ASET_FAIL_UNREC, SUNLS_PACKAGE_FAIL_UNREC,

If the KINSPILS solve function failed (KINSol returned KIN_LSOLVE_FAIL), lsflag contains the error return flag from the SUNLINSOL object, which will be one of the following:
SUNLS_MEM_NULL, indicating that the SUNLINSOL memory is NULL;
SUNLS_ATIMES_FAIL_UNREC, indicating an unrecoverable failure in the Jacobian-times-vector function;
SUNLS_PSOLVE_FAIL_UNREC, indicating that the preconditioner solve function, psolve, failed with an unrecoverable error;
SUNLS_GS_FAIL, indicating a failure in the Gram-Schmidt procedure (generated only in SPGMR or SPFGMR);
SUNLS_QRSOL_FAIL, indicating that the matrix R was found to be singular during the QR solve phase (SPGMR and SPFGMR only); or
SUNLS_PACKAGE_FAIL_UNREC, indicating an unrecoverable failure in an external iterative linear solver package.

KINSpilsGetReturnFlagName

Call `name = KINSpilsGetReturnFlagName(lsflag);`

Description The function KINSpilsGetReturnFlagName returns the name of the KINSPILS constant corresponding to lsflag.

Arguments The only argument, of type `long int`, is a return flag from an KINSPILS function.

Return value The return value is a string containing the name of the corresponding constant.

4.6 User-supplied functions

The user-supplied functions consist of one function defining the nonlinear system, (optionally) a function that handles error and warning messages, (optionally) a function that handles informational messages, (optionally) one or two functions that provides Jacobian-related information for the linear solver, and (optionally) one or two functions that define the preconditioner for use in any of the Krylov iterative algorithms.

4.6.1 Problem-defining function

The user must provide a function of type KINSysFn defined as follows:

KINSysFn

Definition `typedef int (*KINSysFn)(N_Vector u, N_Vector fval, void *user_data);`

Purpose This function computes $F(u)$ (or $G(u)$ for fixed-point iteration and Anderson acceleration) for a given value of the vector u.

Arguments
- `u` is the current value of the variable vector, u.
- `fval` is the output vector $F(u)$.
- `user_data` is a pointer to user data, the pointer `user_data` passed to KINSetUserData.

Return value A KINSysFn function should return 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if it failed unrecoverably (in which case the solution process is halted and KIN_SYSFUNC_FAIL is returned).

Notes Allocation of memory for fval is handled within KINSOL.

4.6.2 Error message handler function

As an alternative to the default behavior of directing error and warning messages to the file pointed to by `errfp` (see `KINSetErrFile`), the user may provide a function of type `KINErrHandlerFn` to process any such messages. The function type `KINErrHandlerFn` is defined as follows:

`KINErrHandlerFn`

Definition
```
typedef void (*KINErrHandlerFn)(int error_code, const char *module,
                                const char *function, char *msg,
                                void *eh_data);
```

Purpose This function processes error and warning messages from KINSOL and its sub-modules.

Arguments
- `error_code` is the error code.
- `module` is the name of the KINSOL module reporting the error.
- `function` is the name of the function in which the error occurred.
- `msg` is the error message.
- `eh_data` is a pointer to user data, the same as the `eh_data` parameter passed to `KINSetErrHandlerFn`.

Return value A `KINErrHandlerFn` function has no return value.

Notes `error_code` is negative for errors and positive (`KIN_WARNING`) for warnings. If a function that returns a pointer to memory encounters an error, it sets `error_code` to 0.

4.6.3 Informational message handler function

As an alternative to the default behavior of directing informational (meaning non-error) messages to the file pointed to by `infofp` (see `KINSetInfoFile`), the user may provide a function of type `KINInfoHandlerFn` to process any such messages. The function type `KINInfoHandlerFn` is defined as follows:

`KINInfoHandlerFn`

Definition
```
typedef void (*KINInfoHandlerFn)(const char *module,
                                 const char *function, char *msg,
                                 void *ih_data);
```

Purpose This function processes informational messages from KINSOL and its sub-modules.

Arguments
- `module` is the name of the KINSOL module reporting the information.
- `function` is the name of the function reporting the information.
- `msg` is the message.
- `ih_data` is a pointer to user data, the same as the `ih_data` parameter passed to `KINSetInfoHandlerFn`.

Return value A `KINInfoHandlerFn` function has no return value.

4.6.4 Jacobian information (direct method Jacobian)

If the direct linear solver interface is used (i.e. (`KINDlsSetLinearSolver` is called in teh step describd in §4.4), the user may provide a function of type `KINDlsJacFn` defined as follows

`KINDlsJacFn`

Definition
```
typedef int (*KINDlsJacFn)(N_Vector u, N_Vector fu,
                           SUNMatrix J, void *user_data,
                           N_Vector tmp1, N_Vector tmp2);
```

Purpose This function computes the Jacobian matrix $J(u)$ or an approximation to it.

Arguments
- `u` is the current (unscaled) iterate.
- `fu` is the current value of the vector $F(u)$.
- `J` is the output approximate Jacobian matrix, $J = \partial F/\partial u$, of type `SUNMatrix`.
- `user_data` is a pointer to user data, the same as the `user_data` parameter passed to `KINSetUserData`.
- `tmp1`
- `tmp2` are pointers to memory allocated for variables of type `N_Vector` which can be used by the `KINJacFn` function as temporary storage or work space.

Return value A function of type `KINDlsDenseJacFn` should return 0 if successful or a non-zero value otherwise.

Notes Information regarding the structure of the specific SUNMATRIX structure (e.g. number of rows, upper/lower bandwidth, sparsity type) may be obtained through using the implementation-specific SUNMATRIX interface functions (see Chapter 7 for details).

Prior to calling the user-supplied Jacobian function, the Jacobian matrix $J(t, y)$ is zeroed out, so only nonzero elements need to be loaded into `Jac`.

If the user's `KINDlsJacFn` function uses difference quotient approximations, it may need to access quantities not in the call list. These quantities may include the scale vectors and the unit roundoff. To obtain the scale vectors, the user will need to add to `user_data` pointers to `u_scale` and/or `f_scale` as needed. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

dense:
A user-supplied dense Jacobian function must load the `N` × `N` dense matrix `Jac` with an approximation to the Jacobian matrix $J(u)$ at the point (`u`). The accessor macros `SM_ELEMENT_D` and `SM_COLUMN_D` allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the SUNMATRIX_DENSE type. `SM_ELEMENT_D(J, i, j)` references the (`i`, `j`)-th element of the dense matrix `Jac` (with `i`, `j` $= 0 \ldots$ `N` $- 1$). This macro is meant for small problems for which efficiency of access is not a major concern. Thus, in terms of the indices m and n ranging from 1 to N, the Jacobian element $J_{m,n}$ can be set using the statement `SM_ELEMENT_D(J, m-1, n-1)` $= J_{m,n}$. Alternatively, `SM_COLUMN_D(J, j)` returns a pointer to the first element of the `j`-th column of `Jac` (with `j` $= 0 \ldots$ `N` $- 1$), and the elements of the `j`-th column can then be accessed using ordinary array indexing. Consequently, $J_{m,n}$ can be loaded using the statements `col_n = SM_COLUMN_D(J, n-1);` `col_n[m-1]` $= J_{m,n}$. For large problems, it is more efficient to use `SM_COLUMN_D` than to use `SM_ELEMENT_D`. Note that both of these macros number rows and columns starting from 0. The SUNMATRIX_DENSE type and accessor macros are documented in §7.1.

banded:
A user-supplied banded Jacobian function must load the `N` × `N` banded matrix `Jac` with an approximation to the Jacobian matrix $J(u)$ at the point (`u`). The accessor macros `SM_ELEMENT_B`, `SM_COLUMN_B`, and `SM_COLUMN_ELEMENT_B` allow the user to read and write banded matrix elements without making specific references to the underlying representation of the SUNMATRIX_BAND type. `SM_ELEMENT_B(J, i, j)` references the (`i`, `j`)-th element of the banded matrix `Jac`, counting from 0. This macro is meant for use in small problems for which efficiency of access is not a major concern. Thus, in terms of the indices m and n ranging from 1 to `N` with (m, n) within the band defined by `mupper` and `mlower`, the Jacobian element $J_{m,n}$ can be loaded using the statement `SM_ELEMENT_B(J, m-1, n-1)` $= J_{m,n}$. The elements within the band are those with `-mupper` $\le$ `m-n` $\le$ `mlower`. Alternatively, `SM_COLUMN_B(J, j)` returns a pointer to the diagonal element of the `j`-th column of `Jac`, and if we assign this address to `realtype *col_j`, then the `i`-th element of the `j`-th column is given by `SM_COLUMN_ELEMENT_B(col_j, i, j)`, counting from 0. Thus, for (m, n) within the band, $J_{m,n}$ can be loaded by setting `col_n`

= SM_COLUMN_B(J, n-1); and SM_COLUMN_ELEMENT_B(col_n, m-1, n-1) = $J_{m,n}$. The elements of the j-th column can also be accessed via ordinary array indexing, but this approach requires knowledge of the underlying storage for a band matrix of type SUNMATRIX_BAND. The array col_n can be indexed from −mupper to mlower. For large problems, it is more efficient to use SM_COLUMN_B and SM_COLUMN_ELEMENT_B than to use the SM_ELEMENT_B macro. As in the dense case, these macros all number rows and columns starting from 0. The SUNMATRIX_BAND type and accessor macros are documented in §7.2.

sparse:
A user-supplied sparse Jacobian function must load the N × N compressed-sparse-column or compressed-sparse-row matrix Jac with an approximation to the Jacobian matrix $J(u)$ at the point (u). Storage for Jac already exists on entry to this function, although the user should ensure that sufficient space is allocated in Jac to hold the nonzero values to be set; if the existing space is insufficient the user may reallocate the data and index arrays as needed. The amount of allocated space in a SUNMATRIX_SPARSE object may be accessed using the macro SM_NNZ_S or the routine SUNSparseMatrix_NNZ. The SUNMATRIX_SPARSE type and accessor macros are documented in §7.3.

4.6.5 Jacobian information (matrix-vector product)

If the KINSPILS solver interface is selected (i.e., KINSpilsSetLinearSolver is called in the steps described in §4.4), the user may provide a function of type KINSpilsJacTimesVecFn in the following form, to compute products Jv. If such a function is not supplied, the default is a difference quotient approximation of these products.

KINSpilsJacTimesVecFn

Definition
```
typedef int (*KINSpilsJacTimesVecFn)(N_Vector v, N_Vector Jv,
                                     N_Vector u, booleantype new_u,
                                     void *user_data);
```

Purpose This jtimes function computes the product Jv (or an approximation to it).

Arguments
- v is the vector by which the Jacobian must be multiplied to the right.
- Jv is the computed output vector.
- u is the current value of the dependent variable vector.
- new_u is a flag, input from KINSOL and possibly reset by the user's jtimes function, indicating whether the iterate vector u has been updated since the last call to jtimes. This is useful if the jtimes function computes and saves Jacobian data that depends on u for use in computing $J(u)v$. The input value of new_u is SUNTRUE following an update by KINSOL, and in that case any saved Jacobian data depending on u should be recomputed. The jtimes routine should then set new_u to SUNFALSE, so that on subsequent calls to jtimes with the same u, the saved data can be reused.
- user_data is a pointer to user data, the same as the user_data parameter passed to KINSetUserData.

Return value The value returned by the Jacobian-times-vector function should be 0 if successful. If a recoverable failure occurred, the return value should be positive. In this case, KINSOL will attempt to correct by calling the preconditioner setup function. If this information is current, KINSOL halts. If the Jacobian-times-vector function encounters an unrecoverable error, it should return a negative value, prompting KINSOL to halt.

Notes If a user-defined routine is not given, then an internal jtimes function, using a difference quotient approximation, is used.

If the user's KINSpilsJacTimesVecFn function uses difference quotient approximations, it may need to access quantities not in the call list. These might include the scale vectors and the unit roundoff. To obtain the scale vectors, the user will need to add to user_data pointers to u_scale and/or f_scale as needed. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundials_types.h.

4.6.6 Preconditioning (linear system solution)

If preconditioning is used, then the user must provide a function to solve the linear system $Pz = r$ where P is the preconditioner matrix, approximating (at least crudely) the system Jacobian $J = \partial F/\partial u$. This function must be of type KINSpilsPrecSolveFn, defined as follows:

KINSpilsPrecSolveFn

Definition
```
typedef int (*KINSpilsPrecSolveFn)(N_Vector u, N_Vector uscale,
                                   N_Vector fval, N_Vector fscale,
                                   N_Vector v, void *user_data);
```

Purpose This function solves the preconditioning system $Pz = r$.

Arguments

u	is the current (unscaled) value of the iterate.
uscale	is a vector containing diagonal elements of the scaling matrix for u.
fval	is the vector $F(u)$ evaluated at u.
fscale	is a vector containing diagonal elements of the scaling matrix for fval.
v	on input, v is set to the right-hand side vector of the linear system, r. On output, v must contain the solution z of the linear system $Pz = r$.
user_data	is a pointer to user data, the same as the user_data parameter passed to the function KINSetUserData.

Return value The value to be returned by the preconditioner solve function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error, and negative for an unrecoverable error.

Notes If the preconditioner solve function fails recoverably and if the preconditioner information (set by the preconditioner setup function) is out of date, KINSOL attempts to correct by calling the setup function. If the preconditioner data is current, KINSOL halts.

4.6.7 Preconditioning (Jacobian data)

If the user's preconditioner requires that any Jacobian-related data be evaluated or preprocessed, then this needs to be done in a user-supplied function of type KINSpilsPrecSetupFn, defined as follows:

KINSpilsPrecSetupFn

Definition
```
typedef int (*KINSpilsPrecSetupFn)(N_Vector u, N_Vector uscale,
                                   N_Vector fval, N_Vector fscale,
                                   void *user_data);
```

Purpose This function evaluates and/or preprocesses Jacobian-related data needed by the preconditioner solve function.

Arguments

u	is the current (unscaled) value of the iterate.
uscale	is a vector containing diagonal elements of the scaling matrix for u.
fval	is the vector $F(u)$ evaluated at u.
fscale	is a vector containing diagonal elements of the scaling matrix for fval.
user_data	is a pointer to user data, the same as the user_data parameter passed to the function KINSetUserData.

Return value The value to be returned by the preconditioner setup function is a flag indicating whether it was successful. This value should be 0 if successful, any other value resulting in halting the KINSOL solver.

Notes The user-supplied preconditioner setup subroutine should compute the right preconditioner matrix P (stored in the memory block referenced by the `user_data` pointer) used to form the scaled preconditioned linear system

$$(D_F J(u) P^{-1} D_u^{-1}) \cdot (D_u P x) = -D_F F(u)\,,$$

where D_u and D_F denote the diagonal scaling matrices whose diagonal elements are stored in the vectors `uscale` and `fscale`, respectively.

The preconditioner setup routine will not be called prior to every call made to the preconditioner solve function, but will instead be called only as often as necessary to achieve convergence of the Newton iteration.

If the user's `KINSpilsPrecSetupFn` function uses difference quotient approximations, it may need to access quantities not in the call list. These might include the scale vectors and the unit roundoff. To obtain the scale vectors, the user will need to add to `user_data` pointers to `u_scale` and/or `f_scale` as needed. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

If the preconditioner solve routine requires no preparation, then a preconditioner setup function need not be given.

4.7 A parallel band-block-diagonal preconditioner module

The efficiency of Krylov iterative methods for the solution of linear systems can be greatly enhanced through preconditioning. For problems in which the user cannot define a more effective, problem-specific preconditioner, KINSOL provides a band-block-diagonal preconditioner module KINBBDPRE, to be used with the parallel `N_Vector` module described in §6.2.

This module provides a preconditioner matrix for KINSOL that is block-diagonal with banded blocks. The blocking corresponds to the distribution of the dependent variable vector u amongst the processes. Each preconditioner block is generated from the Jacobian of the local part (associated with the current process) of a given function $G(u)$ approximating $F(u)$ ($G = F$ is allowed). The blocks are generated by each process via a difference quotient scheme, utilizing a specified band structure. This structure is given by upper and lower half-bandwidths, `mudq` and `mldq`, defined as the number of non-zero diagonals above and below the main diagonal, respectively. However, from the resulting approximate Jacobain blocks, only a matrix of bandwidth `mukeep` + `mlkeep` +1 is retained.

Neither pair of parameters need be the true half-bandwidths of the Jacobian of the local block of G, if smaller values provide a more efficient preconditioner. Such an efficiency gain may occur if the couplings in the system outside a certain bandwidth are considerably weaker than those within the band. Reducing `mukeep` and `mlkeep` while keeping `mudq` and `mldq` at their true values, discards the elements outside the narrower band. Reducing both pairs has the additional effect of lumping the outer Jacobian elements into the computed elements within the band, and requires more caution and experimentation to see whether the lower cost of narrower band matrices offsets the loss of accuracy in the blocks.

The KINBBDPRE module calls two user-provided functions to construct P: a required function `Gloc` (of type `KINBBDLocalFn`) which approximates the nonlinear system function function $G(u) \approx F(u)$ and which is computed locally, and an optional function `Gcomm` (of type `KINBBDCommFn`) which performs all interprocess communication necessary to evaluate the approximate function G. These are in addition to the user-supplied nonlinear system function that evaluates $F(u)$. Both functions take as input the same pointer `user_data` as that passed by the user to `KINSetUserData` and passed to the user's function `func`, and neither function has a return value. The user is responsible for providing space (presumably within `user_data`) for components of `u` that are communicated by `Gcomm` from the other processes, and that are then used by `Gloc`, which should not do any communication.

KINBBDLocalFn

Definition `typedef void (*KINBBDLocalFn)(sunindextype Nlocal, N_Vector u, N_Vector gval, void *user_data);`

Purpose This `Gloc` function computes $G(u)$, and outputs the resulting vector as `gval`.

Arguments `Nlocal` is the local vector length.

`u` is the current value of the iterate.

`gval` is the output vector.

`user_data` is a pointer to user data, the same as the `user_data` parameter passed to `KINSetUserData`.

Return value A `KINBBDLocalFn` function type does not have a return value.

Notes This function must assume that all interprocess communication of data needed to calculate `gval` has already been done, and this data is accessible within `user_data`.

Memory for `u` and `gval` is handled within the preconditioner module.

The case where G is mathematically identical to F is allowed.

KINBBDCommFn

Definition `typedef void (*KINBBDCommFn)(sunindextype Nlocal, N_Vector u, void *user_data);`

Purpose This `Gcomm` function performs all interprocess communications necessary for the execution of the `Gloc` function above, using the input vector `u`.

Arguments `Nlocal` is the local vector length.

`u` is the current value of the iterate.

`user_data` is a pointer to user data, the same as the `user_data` parameter passed to `KINSetUserData`.

Return value A `KINBBDCommFn` function type does not have a return value.

Notes The `Gcomm` function is expected to save communicated data in space defined within the structure `user_data`.

Each call to the `Gcomm` function is preceded by a call to the system function `func` with the same `u` argument. Thus `Gcomm` can omit any communications done by `func` if relevant to the evaluation of `Gloc`. If all necessary communication was done in `func`, then `Gcomm = NULL` can be passed in the call to `KINBBDPrecInit` (see below).

Besides the header files required for the solution of a nonlinear problem (see §4.3), to use the KINBBDPRE module, the main program must include the header file `kinbbdpre.h` which declares the needed function prototypes.

The following is a summary of the usage of this module and describes the sequence of calls in the user main program. Steps that are unchanged from the user main program presented in §4.4 are grayed out.

1. **Initialize parallel or multi-threaded environment**
2. **Set problem dimensions, etc.**
3. **Set vector with initial guess**
4. **Create KINSOL object**
5. **Initialize KINSOL solver**
6. **Set optional inputs**

7. **Create linear solver object**

   When creating the iterative linear solver object, specify use of right preconditioning (`PREC_RIGHT`) as KINSOL only supports right preconditioning.

8. **Attach iterative linear solver module**

9. **Initialize the KINBBDPRE preconditioner module**

   Specify the upper and lower half-bandwidth pairs (`mudq`, `mldq`) and (`mukeep`, `mlkeep`), and call

   ```
   flag = KINBBDPrecInit(kin_mem, Nlocal, mudq, mldq,
                         mukeep, mlkeep, dq_rel_u, Gloc, Gcomm);
   ```

   to allocate memory for and initialize the internal preconditoner data. The last two arguments of `KINBBDPrecInit` are the two user-supplied functions described above.

10. **Set linear solver interface optional inputs**

    Note that the user should not overwrite the preconditioner data, setup function, or solve function through calls to `KINSpilsSetPreconditioner` optional input functions.

11. **Solve problem**

12. **Get optional output**

    Additional optional outputs associated with KINBBDPRE are available by way of two routines described below, `KINBBDPrecGetWorkSpace` and `KINBBDPrecGetNumGfnEvals`.

13. **Deallocate memory for solution vector**

14. **Free solver memory**

15. **Free linear solver memory**

16. **Finalize MPI, if used**

The user-callable function that initializes KINBBDPRE (step 9), is described in more detail below.

`KINBBDPrecInit`

Call
```
flag = KINBBDPrecInit(kin_mem, Nlocal, mudq, mldq,
                      mukeep, mlkeep, dq_rel_u, Gloc, Gcomm);
```

Description The function `KINBBDPrecInit` initializes and allocates memory for the KINBBDPRE preconditioner.

Arguments

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`Nlocal`	(`sunindextype`) local vector length.
`mudq`	(`sunindextype`) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
`mldq`	(`sunindextype`) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
`mukeep`	(`sunindextype`) upper half-bandwidth of the retained banded approximate Jacobian block.
`mlkeep`	(`sunindextype`) lower half-bandwidth of the retained banded approximate Jacobian block.
`dq_rel_u`	(`realtype`) the relative increment in components of `u` used in the difference quotient approximations. The default is `dq_rel_u`$= \sqrt{\text{unit roundoff}}$, which can be specified by passing `dq_rel_u`$= 0.0$.

`Gloc` (`KINBBDLocalFn`) the C function which computes the approximation $G(u) \approx F(u)$.

`Gcomm` (`KINBBDCommFn`) the optional C function which performs all interprocess communication required for the computation of $G(u)$.

Return value The return value `flag` (of type `int`) is one of

`KINSPILS_SUCCESS` The call to `KINBBDPrecInit` was successful.

`KINSPILS_MEM_NULL` The `kin_mem` pointer was `NULL`.

`KINSPILS_MEM_FAIL` A memory allocation request has failed.

`KINSPILS_LMEM_NULL` A KINSPILS linear solver was not attached.

`KINSPILS_ILL_INPUT` The supplied vector implementation was not compatible with the block band preconditioner.

Notes If one of the half-bandwidths `mudq` or `mldq` to be used in the difference-quotient calculation of the approximate Jacobian is negative or exceeds the value `Nlocal`-1, it is replaced with 0 or `Nlocal`-1 accordingly.

The half-bandwidths `mudq` and `mldq` need not be the true half-bandwidths of the Jacobian of the local block of G, when smaller values may provide greater efficiency.

Also, the half-bandwidths `mukeep` and `mlkeep` of the retained banded approximate Jacobian block may be even smaller, to reduce storage and computation costs further.

For all four half-bandwidths, the values need not be the same for every process.

The following two optional output functions are available for use with the KINBBDPRE module:

`KINBBDPrecGetWorkSpace`

Call `flag = KINBBDPrecGetWorkSpace(kin_mem, &lenrwBBDP, &leniwBBDP);`

Description The function `KINBBDPrecGetWorkSpace` returns the local KINBBDPRE real and integer workspace sizes.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`lenrwBBDP` (`long int`) local number of `realtype` values in the KINBBDPRE workspace.

`leniwBBDP` (`long int`) local number of integer values in the KINBBDPRE workspace.

Return value The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS` The optional output value has been successfully set.

`KINSPILS_MEM_NULL` The `kin_mem` pointer was `NULL`.

`KINSPILS_PMEM_NULL` The KINBBDPRE preconditioner has not been initialized.

Notes The workspace requirements reported by this routine correspond only to memory allocated within the KINBBDPRE module (the banded matrix approximation, banded SUNLINSOL object, temporary vectors). These values are local to each process.

The workspaces referred to here exist in addition to those given by the corresponding `KINSp*GetWorkSpace` function.

`KINBBDPrecGetNumGfnEvals`

Call `flag = KINBBDPrecGetNumGfnEvals(kin_mem, &ngevalsBBDP);`

Description The function `KINBBDPrecGetNumGfnEvals` returns the number of calls to the user `Gloc` function due to the difference quotient approximation of the Jacobian blocks used within KINBBDPRE's preconditioner setup function.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`ngevalsBBDP` (`long int`) the number of calls to the user `Gloc` function.

Return value The return value `flag` (of type `int`) is one of:

KINSPILS_SUCCESS	The optional output value has been successfully set.
KINSPILS_MEM_NULL	The kin_mem pointer was NULL.
KINSPILS_PMEM_NULL	The KINBBDPRE preconditioner has not been initialized.

In addition to the ngevalsBBDP Gloc evaluations, the costs associated with KINBBDPRE also include nlinsetups LU factorizations, nlinsetups calls to Gcomm, npsolves banded backsolve calls, and nfevalsLS right-hand side function evaluations, where nlinsetups is an optional KINSOL output and npsolves and nfevalsLS are linear solver optional outputs (see §4.5.5).

Chapter 5

FKINSOL, an Interface Module for FORTRAN Applications

The FKINSOL interface module is a package of C functions which support the use of the KINSOL solver, for the solution of nonlinear systems $F(u) = 0$, in a mixed FORTRAN/C setting. While KINSOL is written in C, it is assumed here that the user's calling program and user-supplied problem-defining routines are written in FORTRAN. This package provides the necessary interface to KINSOL for all supplied serial and parallel NVECTOR implementations.

5.1 Important note on portability

In this package, the names of the interface functions, and the names of the FORTRAN user routines called by them, appear as dummy names which are mapped to actual values by a series of definitions in the header files. By default, those mapping definitions depend in turn on the C macro `F77_FUNC` defined in the header file `sundials_config.h`. The mapping defined by `F77_FUNC` in turn transforms the C interface names to match the name-mangling approach used by the supplied Fortran compiler.

By "name-mangling", we mean that due to the case-independent nature of the FORTRAN language, FORTRAN compilers convert all subroutine and object names to use either all lower-case or all upper-case characters, and append either zero, one or two underscores as a prefix or suffix to the name. For example, the FORTRAN subroutine `MyFunction()` will be changed to one of `myfunction`, `MYFUNCTION`, `myfunction__`, `MYFUNCTION_`, and so on, depending on the FORTRAN compiler used.

SUNDIALS determines this name-mangling scheme at configuration time (see Appendix A).

5.2 Fortran Data Types

Throughout this documentation, we will refer to data types according to their usage in C. The equivalent types to these may vary, depending on your computer architecture and on how SUNDIALS was compiled (see Appendix A). A FORTRAN user should first determine the equivalent types for their architecture and compiler, and then take care that all arguments passed through this FORTRAN/C interface are declared of the appropriate type.

Integers: While SUNDIALS uses the configurable `sunindextype` type as the integer type for vector and matrix indices for its C code, the FORTRAN interfaces are more restricted. The `sunindextype` is only used for index values and pointers when filling sparse matrices. As for C, the `sunindextype` can be configured to be a 32- or 64-bit signed integer by setting the variable `SUNDIALS_INDEX_TYPE` at compile time (See Appendix A). The default value is `int64_t`. A FORTRAN user should set this variable based on the integer type used for vector and matrix indices in their FORTRAN code. The corresponding FORTRAN types are:

- `int32_t` – equivalent to an `INTEGER` or `INTEGER*4` in FORTRAN

- `int64_t` – equivalent to an `INTEGER*8` in FORTRAN

In general, for the FORTRAN interfaces in SUNDIALS, flags of type `int`, vector and matrix lengths, counters, and arguments to `*SETIN()` functions all have `long int` type, and `sunindextype` is only used for index values and pointers when filling sparse matrices. Note that if an F90 (or higher) user wants to find out the value of `sunindextype`, they can include `sundials_fconfig.h`.

Real numbers: As discussed in Appendix A, at compilation SUNDIALS allows the configuration option `SUNDIALS_PRECISION`, that accepts values of `single`, `double` or `extended` (the default is `double`). This choice dictates the size of a `realtype` variable. The corresponding FORTRAN types for these `realtype` sizes are:

- `single` – equivalent to a `REAL` or `REAL*4` in FORTRAN
- `double` – equivalent to a `DOUBLE PRECISION` or `REAL*8` in FORTRAN
- `extended` – equivalent to a `REAL*16` in FORTRAN

5.3 FKINSOL routines

The user-callable functions, with the corresponding KINSOL functions, are as follows:

- Interface to the NVECTOR modules
  - `FNVINITS` (defined by NVECTOR_SERIAL) interfaces to `N_VNewEmpty_Serial`.
  - `FNVINITP` (defined by NVECTOR_PARALLEL) interfaces to `N_VNewEmpty_Parallel`.
  - `FNVINITOMP` (defined by NVECTOR_OPENMP) interfaces to `N_VNewEmpty_OpenMP`.
  - `FNVINITPTS` (defined by NVECTOR_PTHREADS) interfaces to `N_VNewEmpty_Pthreads`.
- Interface to the SUNMATRIX modules
  - `FSUNBANDMATINIT` (defined by SUNMATRIX_BAND) interfaces to `SUNBandMatrix`.
  - `FSUNDENSEMATINIT` (defined by SUNMATRIX_DENSE) interfaces to `SUNDenseMatrix`.
  - `FSUNSPARSEMATINIT` (defined by SUNMATRIX_SPARSE) interfaces to `SUNSparseMatrix`.
- Interface to the SUNLINSOL modules
  - `FSUNBANDLINSOLINIT` (defined by SUNLINSOL_BAND) interfaces to `SUNBandLinearSolver`.
  - `FSUNDENSELINSOLINIT` (defined by SUNLINSOL_DENSE) interfaces to `SUNDenseLinearSolver`.
  - `FSUNKLUINIT` (defined by SUNLINSOL_KLU) interfaces to `SUNKLU`.
  - `FSUNKLUREINIT` (defined by SUNLINSOL_KLU) interfaces to `SUNKLUReinit`.
  - `FSUNLAPACKBANDINIT` (defined by SUNLINSOL_LAPACKBAND) interfaces to `SUNLapackBand`.
  - `FSUNLAPACKDENSEINIT` (defined by SUNLINSOL_LAPACKDENSE) interfaces to `SUNLapackDense`.
  - `FSUNPCGINIT` (defined by SUNLINSOL_PCG) interfaces to `SUNPCG`.
  - `FSUNSPBCGSINIT` (defined by SUNLINSOL_SPBCGS) interfaces to `SUNSPBCGS`.
  - `FSUNSPFGMRINIT` (defined by SUNLINSOL_SPFGMR) interfaces to `SUNSPFGMR`.
  - `FSUNSPGMRINIT` (defined by SUNLINSOL_SPGMR) interfaces to `SUNSPGMR`.
  - `FSUNSPTFQMRINIT` (defined by SUNLINSOL_SPTFQMR) interfaces to `SUNSPTFQMR`.
  - `FSUNSUPERLUMTINIT` (defined by SUNLINSOL_SUPERLUMT) interfaces to `SUNSuperLUMT`.
- Interface to the main KINSOL module

  - FKINCREATE interfaces to KINCreate.
  - FKINSETIIN and FKINSETRIN interface to KINSet* functions.
  - FKININIT interfaces to KINInit.
  - FKINSETVIN interfaces to KINSetConstraints.
  - FKINSOL interfaces to KINSol, KINGet* functions, and to the optional output functions for the selected linear solver module.
  - FKINFREE interfaces to KINFree.

- Interface to the linear solver interfaces
  - FKINDLSINIT interfaces to KINDlsSetLinearSolver.
  - FKINDENSESETJAC interfaces to KINDlsSetJacFn.
  - FKINBANDSETJAC interfaces to KINDlsSetJacFn.
  - FKINSPARSESETJAC interfaces to KINSlsSetJacFn.
  - FKINSPILSINIT interfaces to KINSpilsSetLinearSolver
  - FKINSPILSSETEPSLIN interfaces to KINSpilsSetEpsLin
  - FKINSPILSSETJAC interfaces to KINSpilsSetJacTimes.
  - FKINSPILSSETPREC interfaces to KINSpilsSetPreconditioner.

The user-supplied functions, each listed with the corresponding internal interface function which calls it (and its type within KINSOL), are as follows:

FKINSOL routine (FORTRAN, user-supplied)	KINSOL function (C, interface)	KINSOL type of interface function
FKFUN	FKINfunc	KINSysFn
FKDJAC	FKINDenseJac	KINDlsJacFn
FKBJAC	FKINBandJac	KINDlsJacFn
FKINSPJAC	FKINSparseJac	KINDlsJacFn
FKPSET	FKINPSet	KINSpilsPrecSetupFn
FKPSOL	FKINPSol	KINSpilsPrecSolveFn
FKJTIMES	FKINJtimes	KINSpilsJacTimesVecFn
FKINJTSETUP	FKINJTSetup	KINSpilsJacTimesSetupFn

In contrast to the case of direct use of KINSOL, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program.

5.4 Usage of the FKINSOL interface module

The usage of FKINSOL requires calls to a few different interface functions, depending on the method options selected, and one or more user-supplied routines which define the problem to be solved. These function calls and user routines are summarized separately below. Some details are omitted, and the user is referred to the description of the corresponding KINSOL functions for information on the arguments of any given user-callable interface routine, or of a given user-supplied function called by an interface function.

1. **Nonlinear system function specification**

   The user must, in all cases, supply the following FORTRAN routine

```
SUBROUTINE FKFUN (U, FVAL, IER)
DIMENSION U(*), FVAL(*)
```

It must set the FVAL array to $F(u)$, the system function, as a function of U = u. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if it failed unrecoverably (in which case the solution process is halted).

2. NVECTOR **module initialization**

   If using one of the NVECTOR modules supplied with SUNDIALS, the user must make a call of the form

   ```
   CALL FNVINIT***(...)
   ```

   in which the name and call sequence are as described in the appropriate section of Chapter 6.

3. SUNMATRIX **module initialization**

   If using a Newton or Picard iteration with the direct SUNLINSOL linear solver module and one of the SUNMATRIX modules supplied with SUNDIALS, the user must make a call of the form

   ```
   CALL FSUN***MATINIT(...)
   ```

   in which the name and call sequence are as described in the appropriate section of Chapter 7. Note that the dense, band, or sparse matrix options are usable only in a serial or multi-threaded environment.

4. SUNLINSOL **module initialization**

   If using a Newton or Picard iteration with one of the SUNLINSOL linear solver modules supplied with SUNDIALS, the user must make a call of the form

   ```
   CALL FSUNBANDLINSOLINIT(...)
   CALL FSUNDENSELINSOLINIT(...)
   CALL FSUNKLUINIT(...)
   CALL FSUNLAPACKBANDINIT(...)
   CALL FSUNLAPACKDENSEINIT(...)
   CALL FSUNPCGINIT(...)
   CALL FSUNSPBCGSINIT(...)
   CALL FSUNSPFGMRINIT(...)
   CALL FSUNSPGMRINIT(...)
   CALL FSUNSPTFQMRINIT(...)
   CALL FSUNSUPERLUMTINIT(...)
   ```

   in which the call sequence is as described in the appropriate section of Chapter 8. Note that the dense, band, or sparse solvers are usable only in a serial or multi-threaded environment.

   Once one of these solvers has been initialized, its solver parameters may be modified using a call to the functions

   ```
   CALL FSUNKLUSETORDERING(...)
   CALL FSUNSUPERLUMTSETORDERING(...)
   CALL FSUNPCGSETPRECTYPE(...)
   CALL FSUNPCGSETMAXL(...)
   CALL FSUNSPBCGSSETPRECTYPE(...)
   CALL FSUNSPBCGSSETMAXL(...)
   CALL FSUNSPFGMRSETGSTYPE(...)
   CALL FSUNSPFGMRSETPRECTYPE(...)
   CALL FSUNSPGMRSETGSTYPE(...)
   ```

```
CALL FSUNSPGMRSETPRECTYPE(...)
CALL FSUNSPTFQMRSETPRECTYPE(...)
CALL FSUNSPTFQMRSETMAXL(...)
```

where again the call sequences are described in the appropriate sections of Chapter 8.

5. **Problem specification**

   To create the main solver memory block, make the following call:

   `FKINCREATE`

Call	`CALL FKINCREATE (IER)`
Description	This function creates the KINSOL memory structure.
Arguments	`None.`
Return value	`IER` is the return completion flag. Values are 0 for successful return and −1 otherwise. See printed message for details in case of failure.
Notes	

6. **Set optional inputs**

   Call `FKINSETIIN`, `FKINSETRIN`, and/or `FKINSETVIN`, to set desired optional inputs, if any. See §5.5 for details.

7. **Solver Initialization**

   To set various problem and solution parameters and allocate internal memory, make the following call:

   `FKININIT`

Call	`CALL FKININIT (IOUT, ROUT, IER)`
Description	This function specifies the optional output arrays, allocates internal memory, and initializes KINSOL.
Arguments	`IOUT` is an integer array for integer optional outputs. `ROUT` is a real array for real optional outputs.
Return value	`IER` is the return completion flag. Values are 0 for successful return and −1 otherwise. See printed message for details in case of failure.
Notes	The user integer data array `IOUT` must be declared as `INTEGER*4` or `INTEGER*8` according to the C type `long int`. The optional outputs associated with the main KINSOL integrator are listed in Table 5.2.

8. **Linear solver interface specification**

   The Newton and Picard solution methods in KINSOL involve the solution of linear systems related to the Jacobian of the nonlinear system. To attach the linear solver (and optionally the matrix) objects initialized in steps 3 and 4 above, the user of FKINSOL must initialize the KINDLS or KINSPILS linear solver interface.

   KINDLS **direct linear slver interface**

   To attach a direct SUNLINSOL object and corresponding SUNMATRIX object to the KINDLS interface, then following calls to initialize the SUNLINSOL and SUNMATRIX objects in steps 3 and 4 above, the user must make the call:

   ```
   CALL FKINDLSINIT (IER)
   ```

where IER is an error return flag which is 0 for success or -1 if a memory allocation failure occurred.

KINDLS **with dense Jacobian matrix**

As an option when using the KINDLS interface with the SUNDIALS linear solvers, SUNLINSOL_DENSE or SUNLINSOL_LAPACKDENSE, the user may supply a routine that computes a dense approximation of the system Jacobian $J = \partial F/\partial u$. If supplied, it must have the following form:

```
SUBROUTINE FKDJAC (NEQ, U, FVAL, DJAC, WK1, WK2, IER)
DIMENSION U(*), FVAL(*), DJAC(NEQ,*), WK1(*), WK2(*)
```

Typically this routine will use only NEQ, U, and DJAC. It must compute the Jacobian and store it columnwise in DJAC. The input arguments U and FVAL contain the current values of u and $F(u)$, respectively. The vectors WK1 and WK2, of length NEQ, are provided as work space for use in FKDJAC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if FKDJAC failed unrecoverably (in which case the solution process is halted). NOTE: The argument NEQ has a type consistent with C type `long int` even in the case when the Lapack dense solver is to be used.

If the FKDJAC routine is provided, then, following the call to FKINDLSINIT, the user must make the call:

```
CALL FKINDENSESETJAC (FLAG, IER)
```

with FLAG $\neq 0$ to specify use of the user-supplied Jacobian approximation. The argument IER is an error return flag which is 0 for success or non-zero if an error occurred.

KINDLS **with band Jacobian matrix** As an option when using the KINDLS interface with SUNLINSOL_BAND or SUNLINSOL_LAPACKBAND linear solvers, the user may supply a routine that computes a band approximation of the system Jacobian $J = \partial F/\partial u$. If supplied, it must have the following form:

```
SUBROUTINE FKBJAC (NEQ, MU, ML, MDIM, U, FVAL, BJAC, WK1, WK2, IER)
DIMENSION U(*), FVAL(*), BJAC(MDIM,*), WK1(*), WK2(*)
```

Typically this routine will use only NEQ, MU, ML, U, and BJAC. It must load the MDIM by N array BJAC with the Jacobian matrix at the current u in band form. Store in BJAC(k, j) the Jacobian element $J_{i,j}$ with $k = i - j +$ MU $+1$ ($k = 1 \cdots$ ML + MU + 1) and $j = 1 \cdots N$. The input arguments U and FVAL contain the current values of u, and $F(u)$, respectively. The vectors WK1 and WK2 of length NEQ are provided as work space for use in FKBJAC. IER is an error return flag, which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if FKBJAC failed unrecoverably (in which case the solution process is halted). NOTE: The arguments NEQ, MU, ML, and MDIM have a type consistent with C type `long int` even in the case when the Lapack band solver is to be used.

If the FKBJAC routine is provided, then, following the call to FKINDLSINIT, the user must make the call:

```
CALL FKINBANDSETJAC (FLAG, IER)
```

with FLAG $\neq 0$ to specify use of the user-supplied Jacobian approximation. The argument IER is an error return flag which is 0 for success or non-zero if an error occurred.

KINDLS **with sparse Jacobian matrix**

If the either of the sparse direct interface packages are used, then the user must supply the FKINSPJAC routine that computes a compressed-sparse-column [or compressed-sparse-row if using

KLU] approximation of the system Jacobian $J = \partial F/\partial u$. If supplied, it must have the following form:

```
 SUBROUTINE FKINSPJAC(Y, FY, N, NNZ, JDATA, JINDEXVALS,
&                     JINDEXPTRS, WK1, WK2, IER)
```

Typically this routine will use only `N, NNZ, JDATA, JINDEXVALS` and `JINDEXPTRS`. It must load the `N` by `N` compressed sparse column [or compressed sparse row] matrix with storage for `NNZ` nonzeros, stored in the arrays `JDATA` (nonzero values), `JINDEXVALS` (row [or column] indices for each nonzero), `JINDEXPTRS` (indices for start of each column [or row]), with the Jacobian matrix at the current (y) in CSC [or CSR] form (see `sunmatrix_sparse.h` for more information). The arguments are `Y`, an array containing state variables; `FY`, an array containing residual values; `N`, the number of matrix rows/columns in the Jacobian; `NNZ`, allocated length of nonzero storage; `JDATA`, nonzero values in the Jacobian (of length `NNZ`); `JINDEXVALS`, row [or column] indices for each nonzero in Jacobian (of length `NNZ`); `JINDEXPTRS`, pointers to each Jacobian column [or row] in the two preceding arrays (of length `N`+1); `WK*`, work arrays containing temporary workspace of same size as `Y`; and `IER`, error return code (0 if successful, > 0 if a recoverable error occurred, or < 0 if an unrecoverable error occurred.)

To indicate that the `FKINSPJAC` routine has been provided, then following the call to `FKINDLSINIT`, the following call must be made

```
CALL FKINSPARSESETJAC (IER)
```

The int return flag `IER` is an error return flag which is 0 for success or nonzero for an error.

KINSPILS **iterative linear solver interface**

To attach an iterative SUNLINSOL object to the KINSPILS interface, following the call to initialize the SUNLINSOL object in step 4 above, the user must make the call:

```
CALL FKINSPILSINIT(IER)
```

`IER` is an error return flag set on 0 on success or -1 if a memory failure occurred.

Optional outputs specific to the KINSPILS case are listed in Table 5.2.

Functions used by KINSPILS

Optional user-supplied routines `FKINJTIMES` and `FKINJTSETUP` (see below), can be provided for Jacobian-vector products. (Note that this routine is required if Picard iteration is being used.) If they are, then, following the call to `FKINSPILSINIT`, the user must make the call:

```
CALL FKINSPILSSETJAC (FLAG, IER)
```

with `FLAG` $\neq 0$ to specify use of the user-supplied Jacobian-times-vector approximation. The argument `IER` is an error return flag which is 0 for success or non-zero if an error occurred.

If preconditioning is to be done, then the user must call

```
CALL FKINSPILSSETPREC (FLAG, IER)
```

with `FLAG` $\neq 0$. The return flag `IER` is 0 if successful, or negative if a memory error occurred. In addition, the user program must include preconditioner routines `FKPSOL` and `FKPSET` (see below).

User-supplied functions for KINSPILS

With treatment of the linear systems by any of the Krylov iterative solvers, there are four optional user-supplied routines — `FKINJTIMES`, `FKINJSETUP`, `FKPSOL`, and `FKPSET`. The specifications for these routines are given below.

As an option when using the KINSPILS linear solver interface, the user may supply a routine that computes the product of the system Jacobian $J = \partial F/\partial u$ and a given vector v. If supplied, it must have the following form:

```
SUBROUTINE FKINJTIMES (V, FJV, NEWU, U, IER)
DIMENSION V(*), FJV(*), U(*)
```

Typically this routine will use only `U`, `V`, and `FJV`. It must compute the product vector Jv, where the vector v is stored in `V`, and store the product in `FJV`. The input argument `U` contains the current value of u. On return, set `IER = 0` if `FKINJTIMES` was successful, and nonzero otherwise. `NEWU` is a flag to indicate if `U` has been changed since the last call; if it has, then `NEWU = 1`, and `FKINJTIMES` should recompute any saved Jacobian data it uses and reset `NEWU` to 0. (See §4.6.5.)

If preconditioning is to be included, the following routine must be supplied, for solution of the preconditioner linear system:

```
SUBROUTINE FKPSOL (U, USCALE, FVAL, FSCALE, VTEM, IER)
DIMENSION U(*), USCALE(*), FVAL(*), FSCALE(*), VTEM(*)
```

Typically this routine will use only `U`, `FVAL`, and `VTEM` It must solve the preconditioned linear system $Pz = r$, where $r =$ `VTEM` is input, and store the solution z in `VTEM` as well. Here P is the right preconditioner. If scaling is being used, the routine supplied must also account for scaling on either coordinate or function value, as given in the arrays `USCALE` and `FSCALE`, respectively.

If the user's preconditioner requires that any Jacobian-related data be evaluated or preprocessed, then the following routine can be used for the evaluation and preprocessing of the preconditioner:

```
SUBROUTINE FKPSET (U, USCALE, FVAL, FSCALE, IER)
DIMENSION U(*), USCALE(*), FVAL(*), FSCALE(*)
```

It must perform any evaluation of Jacobian-related data and preprocessing needed for the solution of the preconditioned linear systems by `FKPSOL`. The variables `U` through `FSCALE` are for use in the preconditioning setup process. Typically, the system function `FKFUN` is called before any calls to `FKPSET`, so that `FVAL` will have been updated. `U` is the current solution iterate. If scaling is being used, `USCALE` and `FSCALE` are available for those operations requiring scaling.

On return, set `IER` $= 0$ if `FKPSET` was successful, or set `IER` $= 1$ if an error occurred.

If the user calls `FKINSPILSSETPREC`, the routine `FKPSET` must be provided, even if it is not needed, and then it should return `IER = 0`.

9. **Problem solution**

Solving the nonlinear system is accomplished by making the following call:

```
CALL FKINSOL (U, GLOBALSTRAT, USCALE, FSCALE, IER)
```

The arguments are as follows. `U` is an array containing the initial guess on input, and the solution on return. `GLOBALSTRAT` is an integer (type `INTEGER`) defining the global strategy choice (0 specifies Inexact Newton, 1 indicates Newton with line search, 2 indicates Picard iteration, and 3 indicates Fixed Point iteration). `USCALE` is an array of scaling factors for the `U` vector. `FSCALE` is an array of scaling factors for the `FVAL` vector. `IER` is an integer completion flag and will have one of the following values: 0 to indicate success, 1 to indicate that the initial guess satisfies $F(u) = 0$ within tolerances, 2 to indicate apparent stalling (small step), or a negative value to indicate an error or failure. These values correspond to the `KINSol` returns (see §4.5.3 and §B.2). The values of the optional outputs are available in `IOPT` and `ROPT` (see Table 5.2).

10. **Memory deallocation**

To free the internal memory created by calls to FKINCREATE, FKININIT, FNVINIT*, FKINDLSINIT, FKINSPILSINIT, and FSUN***MATINIT, make the call

```
CALL FKINFREE
```

5.5 FKINSOL optional input and output

In order to keep the number of user-callable FKINSOL interface routines to a minimum, optional inputs to the KINSOL solver are passed through only three routines: FKINSETIIN for integer optional inputs, FKINSETRIN for real optional inputs, and FKINSETVIN for real vector (array) optional inputs. These functions should be called as follows:

```
CALL FKINSETIIN (KEY, IVAL, IER)
CALL FKINSETRIN (KEY, RVAL, IER)
CALL FKINSETVIN (KEY, VVAL, IER)
```

where KEY is a quoted string indicating which optional input is set, IVAL is the integer input value to be used, RVAL is the real input value to be used, and VVAL is the input real array to be used. IER is an integer return flag which is set to 0 on success and a negative value if a failure occurred. For the legal values of KEY in calls to FKINSETIIN and FKINSETRIN, see Table 5.1. The one legal value of KEY for FKINSETVIN is CONSTR_VEC, for providing the array of inequality constraints to be imposed on the solution, if any. The integer IVAL should be declared in a manner consistent with C type long int.

The optional outputs from the KINSOL solver are accessed not through individual functions, but rather through a pair of arrays, IOUT (integer type) of dimension at least 15, and ROUT (real type) of dimension at least 2. These arrays are owned (and allocated) by the user and are passed as arguments to FKININIT. Table 5.2 lists the entries in these two arrays and specifies the optional variable as well as the KINSOL function which is actually called to extract the optional output.

For more details on the optional inputs and outputs, see §4.5.4 and §4.5.5.

5.6 Usage of the FKINBBD interface to KINBBDPRE

The FKINBBD interface sub-module is a package of C functions which, as part of the FKINSOL interface module, support the use of the KINSOL solver with the parallel NVECTOR_PARALLEL module and the KINBBDPRE preconditioner module (see §4.7), for the solution of nonlinear problems in a mixed FORTRAN/C setting.

The user-callable functions in this package, with the corresponding KINSOL and KINBBDPRE functions, are as follows:

- FKINBBDINIT interfaces to KINBBDPrecInit.
- FKINBBDOPT interfaces to KINBBDPRE optional output functions.

In addition to the FORTRAN right-hand side function FKFUN, the user-supplied functions used by this package, are listed below, each with the corresponding interface function which calls it (and its type within KINBBDPRE or KINSOL):

FKINBBD routine (FORTRAN, user-supplied)	KINSOL function (C, interface)	KINSOL type of interface function
FKLOCFN	FKINgloc	KINBBDLocalFn
FKCOMMF	FKINgcomm	KINBBDCommFn
FKJTIMES	FKINJtimes	KINSpilsJacTimesVecFn

Table 5.1: Keys for setting FKINSOL optional inputs

Integer optional inputs FKINSETIIN

Key	Optional input	Default value
PRNT_LEVEL	Verbosity level of output	0
MAA	Number of prior residuals for Anderson Acceleration	0
MAX_NITERS	Maximum no. of nonlinear iterations	200
ETA_FORM	Form of η coefficient	1 (KIN_ETACHOICE1)
MAX_SETUPS	Maximum no. of iterations without prec. setup	10
MAX_SP_SETUPS	Maximum no. of iterations without residual check	5
NO_INIT_SETUP	No initial preconditioner setup	SUNFALSE
NO_MIN_EPS	Lower bound on ϵ	SUNFALSE
NO_RES_MON	No residual monitoring	SUNFALSE

Real optional inputs (FKINSETRIN)

Key	Optional input	Default value
FNORM_TOL	Function-norm stopping tolerance	uround$^{1/3}$
SSTEP_TOL	Scaled-step stopping tolerance	uround$^{2/3}$
MAX_STEP	Max. scaled length of Newton step	$1000\|D_u u_0\|_2$
RERR_FUNC	Relative error for F.D. Jv	$\sqrt{\text{uround}}$
ETA_CONST	Constant value of η	0.1
ETA_PARAMS	Values of γ and α	0.9 and 2.0
RMON_CONST	Constant value of ω	0.9
RMON_PARAMS	Values of ω_{min} and ω_{max}	0.00001 and 0.9

As with the rest of the FKINSOL routines, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program. Additionally, based on flags discussed above in §5.3, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file `fkinbbd.h`.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in §5.4 are grayed-out.

1. **Nonlinear system function specification**
2. NVECTOR **module initialization**
3. SUNLINSOL **module initialization**

   Initialize one of the iterative SUNLINSOL modules, by calling one of FSUNPCGINIT, FSUNSPBCGSINIT, FSUNSPFGMRINIT, FSUNSPGMRINIT or FSUNSPTFQMRINIT.
4. **Problem specification**
5. **Set optional inputs**
6. KINSOL **Initialization**
7. **Linear solver interface specification**

   Initialize the KINSPILS iterative linear solver interface by calling FKINSPILSINIT.

   To initialize the KINBBDPRE preconditioner, make the following call:

```
CALL FKINBBDINIT (NLOCAL, MUDQ, MLDQ, MU, ML, IER)
```

   The arguments are as follows. NLOCAL is the local size of vectors for this process. MUDQ and MLDQ are the upper and lower half-bandwidths to be used in the computation of the local Jacobian blocks

Table 5.2: Description of the FKINSOL optional output arrays IOUT and ROUT

Integer output array IOUT

Index	Optional output	KINSOL function
KINSOL main solver		
1	LENRW	KINGetWorkSpace
2	LENIW	KINGetWorkSpace
3	NNI	KINGetNumNonlinSolvIters
4	NFE	KINGetNumFuncEvals
5	NBCF	KINGetNumBetaCondFails
6	NBKTRK	KINGetNumBacktrackOps
KINDLS linear solver interface		
7	LENRWLS	KINDlsGetWorkSpace
8	LENIWLS	KINDlsGetWorkSpace
9	LS_FLAG	KINDlsGetLastFlag
10	NFELS	KINDlsGetNumFuncEvals
11	NJE	KINDlsGetNumJacEvals
KINSPILS linear solver interface		
7	LENRWLS	KINSpilsGetWorkSpace
8	LENIWLS	KINSpilsGetWorkSpace
9	LS_FLAG	KINSpilsGetLastFlag
10	NFELS	KINSpilsGetNumFuncEvals
11	NJTV	KINSpilsGetNumJacEvals
12	NPE	KINSpilsGetNumPrecEvals
13	NPS	KINSpilsGetNumPrecSolves
14	NLI	KINSpilsGetNumLinIters
15	NCFL	KINSpilsGetNumConvFails

Real output array ROUT

Index	Optional output	KINSOL function
1	FNORM	KINGetFuncNorm
2	SSTEP	KINGetStepLength

by difference quotients; these may be smaller than the true half-bandwidths of the Jacobian of the local block of G, when smaller values may provide greater efficiency. MU and ML are the upper and lower half-bandwidths of the band matrix that is retained as an approximation of the local Jacobian block; these may be smaller than MUDQ and MLDQ. IER is a return completion flag. A value of 0 indicates success, while a value of -1 indicates that a memory failure occurred or that an input had an illegal value.

Optionally, to specify that the SPGMR, SPFGMR, SPBCGS, or SPTFQMR solver should use the supplied FKJTIMES, make the call

```
CALL FKINSPILSSETJAC (FLAG, IER)
```

with FLAG $\neq 0$. (See step 8 in §5.4).

8. **Problem solution**

9. **Additional solution output**

10. KINBBDPRE **Optional outputs**

Optional outputs specific to the SPGMR, SPFGMR, SPBCGS, or SPTFQMR solver are listed in Table 5.2. To obtain the optional outputs associated with the KINBBDPRE module, make the following call:

```
CALL FKINBBDOPT (LENRBBD, LENIBBD, NGEBBD)
```

The arguments should be consistent with C type `long int`. Their returned values are as follows: LENRBBD is the length of real preconditioner work space, in `realtype` words. LENIBBD is the length of integer preconditioner work space, in integer words. These sizes are local to the current process. NGEBBD is the cumulative number of $G(u)$ evaluations (calls to FKLOCFN) so far.

11. **Memory deallocation**

(The memory allocated for the FKINBBD module is deallocated automatically by FKINFREE.)

12. **User-supplied routines**

The following two routines must be supplied for use with the KINBBDPRE module:

```
SUBROUTINE FKLOCFN (NLOC, ULOC, GLOC, IER)
DIMENSION ULOC(*), GLOC(*)
```

This routine is to evaluate the function $G(u)$ approximating F (possibly identical to F), in terms of the array ULOC (of length NLOC), which is the sub-vector of u local to this processor. The resulting (local) sub-vector is to be stored in the array GLOC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if FKLOCFN failed unrecoverably (in which case the solution process is halted).

```
SUBROUTINE FKCOMMFN (NLOC, ULOC, IER)
DIMENSION ULOC(*)
```

This routine is to perform the inter-processor communication necessary for the FKLOCFN routine. Each call to FKCOMMFN is preceded by a call to the system function routine FKFUN with the same argument ULOC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if FKCOMMFN failed recoverably (in which case the solution process is halted).

The subroutine `FKCOMMFN` must be supplied even if it is not needed and must return `IER = 0`.

Optionally, the user can supply a routine `FKINJTIMES` for the evaluation of Jacobian-vector products, as described above in step 8 in §5.4. Note that this routine is required if using Picard iteration.

Chapter 6

Description of the NVECTOR module

The SUNDIALS solvers are written in a data-independent manner. They all operate on generic vectors (of type `N_Vector`) through a set of operations defined by the particular NVECTOR implementation. Users can provide their own specific implementation of the NVECTOR module, or use one of the implementations provided with SUNDIALS. The generic operations are described below and the implementations provided with SUNDIALS are described in the following sections.

The generic `N_Vector` type is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the vector, and an *ops* field pointing to a structure with generic vector operations. The type `N_Vector` is defined as

```
typedef struct _generic_N_Vector *N_Vector;

struct _generic_N_Vector {
    void *content;
    struct _generic_N_Vector_Ops *ops;
};
```

The `_generic_N_Vector_Ops` structure is essentially a list of pointers to the various actual vector operations, and is defined as

```
struct _generic_N_Vector_Ops {
  N_Vector_ID (*nvgetvectorid)(N_Vector);
  N_Vector    (*nvclone)(N_Vector);
  N_Vector    (*nvcloneempty)(N_Vector);
  void        (*nvdestroy)(N_Vector);
  void        (*nvspace)(N_Vector, sunindextype *, sunindextype *);
  realtype*   (*nvgetarraypointer)(N_Vector);
  void        (*nvsetarraypointer)(realtype *, N_Vector);
  void        (*nvlinearsum)(realtype, N_Vector, realtype, N_Vector, N_Vector);
  void        (*nvconst)(realtype, N_Vector);
  void        (*nvprod)(N_Vector, N_Vector, N_Vector);
  void        (*nvdiv)(N_Vector, N_Vector, N_Vector);
  void        (*nvscale)(realtype, N_Vector, N_Vector);
  void        (*nvabs)(N_Vector, N_Vector);
  void        (*nvinv)(N_Vector, N_Vector);
  void        (*nvaddconst)(N_Vector, realtype, N_Vector);
  realtype    (*nvdotprod)(N_Vector, N_Vector);
  realtype    (*nvmaxnorm)(N_Vector);
  realtype    (*nvwrmsnorm)(N_Vector, N_Vector);
```

```
  realtype    (*nvwrmsnormmask)(N_Vector, N_Vector, N_Vector);
  realtype    (*nvmin)(N_Vector);
  realtype    (*nvwl2norm)(N_Vector, N_Vector);
  realtype    (*nvl1norm)(N_Vector);
  void        (*nvcompare)(realtype, N_Vector, N_Vector);
  booleantype (*nvinvtest)(N_Vector, N_Vector);
  booleantype (*nvconstrmask)(N_Vector, N_Vector, N_Vector);
  realtype    (*nvminquotient)(N_Vector, N_Vector);
};
```

The generic NVECTOR module defines and implements the vector operations acting on `N_Vector`. These routines are nothing but wrappers for the vector operations defined by a particular NVECTOR implementation, which are accessed through the *ops* field of the `N_Vector` structure. To illustrate this point we show below the implementation of a typical vector operation from the generic NVECTOR module, namely `N_VScale`, which performs the scaling of a vector `x` by a scalar `c`:

```
void N_VScale(realtype c, N_Vector x, N_Vector z)
{
   z->ops->nvscale(c, x, z);
}
```

Table 6.2 contains a complete list of all vector operations defined by the generic NVECTOR module.

Finally, note that the generic NVECTOR module defines the functions `N_VCloneVectorArray` and `N_VCloneVectorArrayEmpty`. Both functions create (by cloning) an array of `count` variables of type `N_Vector`, each of the same type as an existing `N_Vector`. Their prototypes are

```
N_Vector *N_VCloneVectorArray(int count, N_Vector w);
N_Vector *N_VCloneVectorArrayEmpty(int count, N_Vector w);
```

and their definitions are based on the implementation-specific `N_VClone` and `N_VCloneEmpty` operations, respectively.

An array of variables of type `N_Vector` can be destroyed by calling `N_VDestroyVectorArray`, whose prototype is

```
void N_VDestroyVectorArray(N_Vector *vs, int count);
```

and whose definition is based on the implementation-specific `N_VDestroy` operation.

A particular implementation of the NVECTOR module must:

- Specify the *content* field of `N_Vector`.
- Define and implement the vector operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one NVECTOR module (each with different `N_Vector` internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free an `N_Vector` with the new *content* field and with *ops* pointing to the new vector operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined `N_Vector` (e.g., a routine to print the content for debugging purposes).
- Optionally, provide accessor macros as needed for that particular implementation to be used to access different parts in the *content* field of the newly defined `N_Vector`.

Each NVECTOR implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 6.1. It is recommended that a user-supplied NVECTOR implementation use the `SUNDIALS_NVEC_CUSTOM` identifier.

Table 6.1: Vector Identifications associated with vector kernels supplied with SUNDIALS.

Vector ID	Vector type	ID Value
SUNDIALS_NVEC_SERIAL	Serial	0
SUNDIALS_NVEC_PARALLEL	Distributed memory parallel (MPI)	1
SUNDIALS_NVEC_OPENMP	OpenMP shared memory parallel	2
SUNDIALS_NVEC_PTHREADS	PThreads shared memory parallel	3
SUNDIALS_NVEC_PARHYP	*hypre* ParHyp parallel vector	4
SUNDIALS_NVEC_PETSC	PETSc parallel vector	5
SUNDIALS_NVEC_CUSTOM	User-provided custom vector	6

Table 6.2: Description of the NVECTOR operations

Name	Usage and Description
`N_VGetVectorID`	`id = N_VGetVectorID(w);` Returns the vector type identifier for the vector `w`. It is used to determine the vector implementation type (e.g. serial, parallel,...) from the abstract `N_Vector` interface. Returned values are given in Table 6.1.
`N_VClone`	`v = N_VClone(w);` Creates a new `N_Vector` of the same type as an existing vector `w` and sets the *ops* field. It does not copy the vector, but rather allocates storage for the new vector.
`N_VCloneEmpty`	`v = N_VCloneEmpty(w);` Creates a new `N_Vector` of the same type as an existing vector `w` and sets the *ops* field. It does not allocate storage for data.
`N_VDestroy`	`N_VDestroy(v);` Destroys the `N_Vector v` and frees memory allocated for its internal data.
`N_VSpace`	`N_VSpace(nvSpec, &lrw, &liw);` Returns storage requirements for one `N_Vector`. `lrw` contains the number of realtype words and `liw` contains the number of integer words. This function is advisory only, for use in determining a user's total space requirements; it could be a dummy function in a user-supplied NVECTOR module if that information is not of interest.

continued on next page

continued from last page

Name	Usage and Description
N_VGetArrayPointer	`vdata = N_VGetArrayPointer(v);` Returns a pointer to a `realtype` array from the `N_Vector v`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realtype`. This routine is only used in the solver-specific interfaces to the dense and banded (serial) linear solvers, the sparse linear solvers (serial and threaded), and in the interfaces to the banded (serial) and band-block-diagonal (parallel) preconditioner modules provided with SUNDIALS.
N_VSetArrayPointer	`N_VSetArrayPointer(vdata, v);` Overwrites the data in an `N_Vector` with a given array of `realtype`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realtype`. This routine is only used in the interfaces to the dense (serial) linear solver, hence need not exist in a user-supplied NVECTOR module for a parallel environment.
N_VLinearSum	`N_VLinearSum(a, x, b, y, z);` Performs the operation $z = ax + by$, where a and b are `realtype` scalars and x and y are of type `N_Vector`: $z_i = ax_i + by_i$, $i = 0, \ldots, n-1$.
N_VConst	`N_VConst(c, z);` Sets all components of the `N_Vector z` to `realtype c`: $z_i = c$, $i = 0, \ldots, n-1$.
N_VProd	`N_VProd(x, y, z);` Sets the `N_Vector z` to be the component-wise product of the `N_Vector` inputs `x` and `y`: $z_i = x_i y_i$, $i = 0, \ldots, n-1$.
N_VDiv	`N_VDiv(x, y, z);` Sets the `N_Vector z` to be the component-wise ratio of the `N_Vector` inputs `x` and `y`: $z_i = x_i/y_i$, $i = 0, \ldots, n-1$. The y_i may not be tested for 0 values. It should only be called with a `y` that is guaranteed to have all nonzero components.
N_VScale	`N_VScale(c, x, z);` Scales the `N_Vector x` by the `realtype` scalar `c` and returns the result in `z`: $z_i = cx_i$, $i = 0, \ldots, n-1$.
N_VAbs	`N_VAbs(x, z);` Sets the components of the `N_Vector z` to be the absolute values of the components of the `N_Vector x`: $y_i = \lvert x_i \rvert$, $i = 0, \ldots, n-1$.

continued on next page

continued from last page

Name	Usage and Description
N_VInv	`N_VInv(x, z);` Sets the components of the `N_Vector z` to be the inverses of the components of the `N_Vector x`: $z_i = 1.0/x_i$, $i = 0, \ldots, n-1$. This routine may not check for division by 0. It should be called only with an `x` which is guaranteed to have all nonzero components.
N_VAddConst	`N_VAddConst(x, b, z);` Adds the `realtype` scalar `b` to all components of `x` and returns the result in the `N_Vector z`: $z_i = x_i + b$, $i = 0, \ldots, n-1$.
N_VDotProd	`d = N_VDotProd(x, y);` Returns the value of the ordinary dot product of `x` and `y`: $d = \sum_{i=0}^{n-1} x_i y_i$.
N_VMaxNorm	`m = N_VMaxNorm(x);` Returns the maximum norm of the `N_Vector x`: $m = \max_i \lvert x_i \rvert$.
N_VWrmsNorm	`m = N_VWrmsNorm(x, w)` Returns the weighted root-mean-square norm of the `N_Vector x` with `realtype` weight vector `w`: $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i)^2\right)/n}$.
N_VWrmsNormMask	`m = N_VWrmsNormMask(x, w, id);` Returns the weighted root mean square norm of the `N_Vector x` with `realtype` weight vector `w` built using only the elements of `x` corresponding to nonzero elements of the `N_Vector id`: $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i \mathrm{sign}(id_i))^2\right)/n}$.
N_VMin	`m = N_VMin(x);` Returns the smallest element of the `N_Vector x`: $m = \min_i x_i$.
N_VWL2Norm	`m = N_VWL2Norm(x, w);` Returns the weighted Euclidean ℓ_2 norm of the `N_Vector x` with `realtype` weight vector `w`: $m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2}$.
N_VL1Norm	`m = N_VL1Norm(x);` Returns the ℓ_1 norm of the `N_Vector x`: $m = \sum_{i=0}^{n-1} \lvert x_i \rvert$.
N_VCompare	`N_VCompare(c, x, z);` Compares the components of the `N_Vector x` to the `realtype` scalar `c` and returns an `N_Vector z` such that: $z_i = 1.0$ if $\lvert x_i \rvert \geq c$ and $z_i = 0.0$ otherwise.

continued on next page

continued from last page	
Name	**Usage and Description**
N_VInvTest	`t = N_VInvTest(x, z);` Sets the components of the `N_Vector z` to be the inverses of the components of the `N_Vector x`, with prior testing for zero values: $z_i = 1.0/x_i$, $i = 0, \ldots, n-1$. This routine returns a boolean assigned to `SUNTRUE` if all components of `x` are nonzero (successful inversion) and returns `SUNFALSE` otherwise.
N_VConstrMask	`t = N_VConstrMask(c, x, m);` Performs the following constraint tests: $x_i > 0$ if $c_i = 2$, $x_i \geq 0$ if $c_i = 1$, $x_i \leq 0$ if $c_i = -1$, $x_i < 0$ if $c_i = -2$. There is no constraint on x_i if $c_i = 0$. This routine returns a boolean assigned to `SUNFALSE` if any element failed the constraint test and assigned to `SUNTRUE` if all passed. It also sets a mask vector `m`, with elements equal to 1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking.
N_VMinQuotient	`minq = N_VMinQuotient(num, denom);` This routine returns the minimum of the quotients obtained by termwise dividing $\texttt{num}_i$ by $\texttt{denom}_i$. A zero element in `denom` will be skipped. If no such quotients are found, then the large value `BIG_REAL` (defined in the header file `sundials_types.h`) is returned.

6.1 The NVECTOR_SERIAL implementation

The serial implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_SERIAL, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, and a boolean flag *own_data* which specifies the ownership of *data*.

```
struct _N_VectorContent_Serial {
  sunindextype length;
  booleantype own_data;
  realtype *data;
};
```

The header file to include when using this module is `nvector_serial.h`. The installed module library to link to is `libsundials_nvecserial.`*`lib`* where *`.lib`* is typically `.so` for shared libraries and `.a` for static libraries.

The following macros are provided to access the content of an NVECTOR_SERIAL vector. The suffix `_S` in the names denotes the serial version.

- `NV_CONTENT_S`

  This routine gives access to the contents of the serial vector `N_Vector`.

  The assignment `v_cont = NV_CONTENT_S(v)` sets `v_cont` to be a pointer to the serial `N_Vector` content structure.

  Implementation:

  ```
  #define NV_CONTENT_S(v) ( (N_VectorContent_Serial)(v->content) )
  ```

- `NV_OWN_DATA_S`, `NV_DATA_S`, `NV_LENGTH_S`

  These macros give individual access to the parts of the content of a serial `N_Vector`.

The assignment `v_data = NV_DATA_S(v)` sets `v_data` to be a pointer to the first component of the data for the `N_Vector` `v`. The assignment `NV_DATA_S(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

The assignment `v_len = NV_LENGTH_S(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_S(v) = len_v` sets the length of `v` to be `len_v`.

Implementation:

```
#define NV_OWN_DATA_S(v) ( NV_CONTENT_S(v)->own_data )
#define NV_DATA_S(v) ( NV_CONTENT_S(v)->data )
#define NV_LENGTH_S(v) ( NV_CONTENT_S(v)->length )
```

- `NV_Ith_S`

  This macro gives access to the individual components of the data array of an `N_Vector`.

  The assignment `r = NV_Ith_S(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_S(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

  Here i ranges from 0 to $n-1$ for a vector of length n.

  Implementation:

  ```
  #define NV_Ith_S(v,i) ( NV_DATA_S(v)[i] )
  ```

The NVECTOR_SERIAL module defines serial implementations of all vector operations listed in Table 6.2. Their names are obtained from those in Table 6.2 by appending the suffix `_Serial` (e.g. `N_VDestroy_Serial`). The module NVECTOR_SERIAL provides the following additional user-callable routines:

- `N_VNew_Serial`

  This function creates and allocates memory for a serial `N_Vector`. Its only argument is the vector length.

  ```
  N_Vector N_VNew_Serial(sunindextype vec_length);
  ```

- `N_VNewEmpty_Serial`

  This function creates a new serial `N_Vector` with an empty (`NULL`) data array.

  ```
  N_Vector N_VNewEmpty_Serial(sunindextype vec_length);
  ```

- `N_VMake_Serial`

  This function creates and allocates memory for a serial vector with user-provided data array.

  (This function does *not* allocate memory for `v_data` itself.)

  ```
  N_Vector N_VMake_Serial(sunindextype vec_length, realtype *v_data);
  ```

- `N_VCloneVectorArray_Serial`

  This function creates (by cloning) an array of `count` serial vectors.

  ```
  N_Vector *N_VCloneVectorArray_Serial(int count, N_Vector w);
  ```

- `N_VCloneVectorArrayEmpty_Serial`

  This function creates (by cloning) an array of `count` serial vectors, each with an empty (`NULL`) data array.

  ```
  N_Vector *N_VCloneVectorArrayEmpty_Serial(int count, N_Vector w);
  ```

- `N_VDestroyVectorArray_Serial`

  This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_Serial` or with `N_VCloneVectorArrayEmpty_Serial`.

  ```
  void N_VDestroyVectorArray_Serial(N_Vector *vs, int count);
  ```

- N_VGetLength_Serial

  This function returns the number of vector elements.

  ```
  sunindextype N_VGetLength_Serial(N_Vector v);
  ```

- N_VPrint_Serial

  This function prints the content of a serial vector to stdout.

  ```
  void N_VPrint_Serial(N_Vector v);
  ```

Notes

- When looping over the components of an N_Vector v, it is more efficient to first obtain the component array via v_data = NV_DATA_S(v) and then access v_data[i] within the loop than it is to use NV_Ith_S(v,i) within the loop.

- N_VNewEmpty_Serial, N_VMake_Serial, and N_VCloneVectorArrayEmpty_Serial set the field *own_data* = SUNFALSE. N_VDestroy_Serial and N_VDestroyVectorArray_Serial will not attempt to free the pointer *data* for any N_Vector with *own_data* set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the *data* pointer.

- To maximize efficiency, vector operations in the NVECTOR_SERIAL implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

For solvers that include a Fortran interface module, the NVECTOR_SERIAL module also includes a Fortran-callable function FNVINITS(code, NEQ, IER), to initialize this NVECTOR_SERIAL module. Here code is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); NEQ is the problem size (declared so as to match C type long int); and IER is an error return flag equal 0 for success and -1 for failure.

6.2 The NVECTOR_PARALLEL implementation

The NVECTOR_PARALLEL implementation of the NVECTOR module provided with SUNDIALS is based on MPI. It defines the *content* field of N_Vector to be a structure containing the global and local lengths of the vector, a pointer to the beginning of a contiguous local data array, an MPI communicator, and a boolean flag *own_data* indicating ownership of the data array *data*.

```
struct _N_VectorContent_Parallel {
  sunindextype local_length;
  sunindextype global_length;
  booleantype own_data;
  realtype *data;
  MPI_Comm comm;
};
```

The header file to include when using this module is nvector_parallel.h. The installed module library to link to is libsundials_nvecparallel.*lib* where *.lib* is typically .so for shared libraries and .a for static libraries.

The following macros are provided to access the content of a NVECTOR_PARALLEL vector. The suffix _P in the names denotes the distributed memory parallel version.

- NV_CONTENT_P

  This macro gives access to the contents of the parallel vector N_Vector.

  The assignment v_cont = NV_CONTENT_P(v) sets v_cont to be a pointer to the N_Vector content structure of type struct _N_VectorContent_Parallel.

Implementation:

```
#define NV_CONTENT_P(v) ( (N_VectorContent_Parallel)(v->content) )
```

- NV_OWN_DATA_P, NV_DATA_P, NV_LOCLENGTH_P, NV_GLOBLENGTH_P

  These macros give individual access to the parts of the content of a parallel N_Vector.

  The assignment v_data = NV_DATA_P(v) sets v_data to be a pointer to the first component of the local data for the N_Vector v. The assignment NV_DATA_P(v) = v_data sets the component array of v to be v_data by storing the pointer v_data.

  The assignment v_llen = NV_LOCLENGTH_P(v) sets v_llen to be the length of the local part of v. The call NV_LENGTH_P(v) = llen_v sets the local length of v to be llen_v.

  The assignment v_glen = NV_GLOBLENGTH_P(v) sets v_glen to be the global length of the vector v. The call NV_GLOBLENGTH_P(v) = glen_v sets the global length of v to be glen_v.

  Implementation:

```
#define NV_OWN_DATA_P(v)    ( NV_CONTENT_P(v)->own_data )
#define NV_DATA_P(v)        ( NV_CONTENT_P(v)->data )
#define NV_LOCLENGTH_P(v)   ( NV_CONTENT_P(v)->local_length )
#define NV_GLOBLENGTH_P(v) ( NV_CONTENT_P(v)->global_length )
```

- NV_COMM_P

  This macro provides access to the MPI communicator used by the NVECTOR_PARALLEL vectors.

  Implementation:

```
#define NV_COMM_P(v) ( NV_CONTENT_P(v)->comm )
```

- NV_Ith_P

  This macro gives access to the individual components of the local data array of an N_Vector.

  The assignment r = NV_Ith_P(v,i) sets r to be the value of the i-th component of the local part of v. The assignment NV_Ith_P(v,i) = r sets the value of the i-th component of the local part of v to be r.

  Here i ranges from 0 to $n-1$, where n is the local length.

  Implementation:

```
#define NV_Ith_P(v,i) ( NV_DATA_P(v)[i] )
```

The NVECTOR_PARALLEL module defines parallel implementations of all vector operations listed in Table 6.2 Their names are obtained from those in Table 6.2 by appending the suffix _Parallel (e.g. N_VDestroy_Parallel). The module NVECTOR_PARALLEL provides the following additional user-callable routines:

- N_VNew_Parallel

  This function creates and allocates memory for a parallel vector.

```
N_Vector N_VNew_Parallel(MPI_Comm comm,
                         sunindextype local_length,
                         sunindextype global_length);
```

- N_VNewEmpty_Parallel

  This function creates a new parallel N_Vector with an empty (NULL) data array.

```
N_Vector N_VNewEmpty_Parallel(MPI_Comm comm,
                              sunindextype local_length,
                              sunindextype global_length);
```

- N_VMake_Parallel

  This function creates and allocates memory for a parallel vector with user-provided data array. (This function does *not* allocate memory for v_data itself.)

```
N_Vector N_VMake_Parallel(MPI_Comm comm,
                          sunindextype local_length,
                          sunindextype global_length,
                          realtype *v_data);
```

- N_VCloneVectorArray_Parallel

  This function creates (by cloning) an array of count parallel vectors.

```
N_Vector *N_VCloneVectorArray_Parallel(int count, N_Vector w);
```

- N_VCloneVectorArrayEmpty_Parallel

  This function creates (by cloning) an array of count parallel vectors, each with an empty (NULL) data array.

```
N_Vector *N_VCloneVectorArrayEmpty_Parallel(int count, N_Vector w);
```

- N_VDestroyVectorArray_Parallel

  This function frees memory allocated for the array of count variables of type N_Vector created with N_VCloneVectorArray_Parallel or with N_VCloneVectorArrayEmpty_Parallel.

```
void N_VDestroyVectorArray_Parallel(N_Vector *vs, int count);
```

- N_VGetLength_Parallel

  This function returns the number of vector elements (global vector length).

```
sunindextype N_VGetLength_Parallel(N_Vector v);
```

- N_VGetLocalLength_Parallel

  This function returns the local vector length.

```
sunindextype N_VGetLocalLength_Parallel(N_Vector v);
```

- N_VPrint_Parallel

  This function prints the content of a parallel vector to stdout.

```
void N_VPrint_Parallel(N_Vector v);
```

Notes

- When looping over the components of an N_Vector v, it is more efficient to first obtain the local component array via v_data = NV_DATA_P(v) and then access v_data[i] within the loop than it is to use NV_Ith_P(v,i) within the loop.

- N_VNewEmpty_Parallel, N_VMake_Parallel, and N_VCloneVectorArrayEmpty_Parallel set the field *own_data* = SUNFALSE. N_VDestroy_Parallel and N_VDestroyVectorArray_Parallel will not attempt to free the pointer *data* for any N_Vector with *own_data* set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the *data* pointer.

- To maximize efficiency, vector operations in the NVECTOR_PARALLEL implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

For solvers that include a Fortran interface module, the NVECTOR_PARALLEL module also includes a Fortran-callable function `FNVINITP(COMM, code, NLOCAL, NGLOBAL, IER)`, to initialize this NVECTOR_PARALLEL module. Here `COMM` is the MPI communicator, `code` is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); `NLOCAL` and `NGLOBAL` are the local and global vector sizes, respectively (declared so as to match C type `long int`); and IER is an error return flag equal 0 for success and -1 for failure. NOTE: If the header file `sundials_config.h` defines `SUNDIALS_MPI_COMM_F2C` to be 1 (meaning the MPI implementation used to build SUNDIALS includes the `MPI_Comm_f2c` function), then `COMM` can be any valid MPI communicator. Otherwise, `MPI_COMM_WORLD` will be used, so just pass an integer value as a placeholder.

6.3 The NVECTOR_OPENMP implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR_OPENMP, and an implementation using Pthreads, called NVECTOR_PTHREADS. Testing has shown that vectors should be of length at least 100,000 before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The OpenMP NVECTOR implementation provided with SUNDIALS, NVECTOR_OPENMP, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag *own_data* which specifies the ownership of *data*, and the number of threads. Operations on the vector are threaded using OpenMP.

```
struct _N_VectorContent_OpenMP {
  sunindextype length;
  booleantype own_data;
  realtype *data;
  int num_threads;
};
```

The header file to include when using this module is `nvector_openmp.h`. The installed module library to link to is `libsundials_nvecopenmp.`*`lib`* where *`.lib`* is typically `.so` for shared libraries and `.a` for static libraries.

The following macros are provided to access the content of an NVECTOR_OPENMP vector. The suffix `_OMP` in the names denotes the OpenMP version.

- `NV_CONTENT_OMP`

  This routine gives access to the contents of the OpenMP vector `N_Vector`.

  The assignment `v_cont = NV_CONTENT_OMP(v)` sets `v_cont` to be a pointer to the OpenMP `N_Vector` content structure.

  Implementation:

  ```
  #define NV_CONTENT_OMP(v) ( (N_VectorContent_OpenMP)(v->content) )
  ```

- `NV_OWN_DATA_OMP`, `NV_DATA_OMP`, `NV_LENGTH_OMP`, `NV_NUM_THREADS_OMP`

  These macros give individual access to the parts of the content of a OpenMP `N_Vector`.

  The assignment `v_data = NV_DATA_OMP(v)` sets `v_data` to be a pointer to the first component of the data for the `N_Vector` `v`. The assignment `NV_DATA_OMP(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

  The assignment `v_len = NV_LENGTH_OMP(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_OMP(v) = len_v` sets the length of `v` to be `len_v`.

  The assignment `v_num_threads = NV_NUM_THREADS_OMP(v)` sets `v_num_threads` to be the number of threads from `v`. On the other hand, the call `NV_NUM_THREADS_OMP(v) = num_threads_v` sets the number of threads for `v` to be `num_threads_v`.

Implementation:

```
#define NV_OWN_DATA_OMP(v) ( NV_CONTENT_OMP(v)->own_data )
#define NV_DATA_OMP(v) ( NV_CONTENT_OMP(v)->data )
#define NV_LENGTH_OMP(v) ( NV_CONTENT_OMP(v)->length )
#define NV_NUM_THREADS_OMP(v) ( NV_CONTENT_OMP(v)->num_threads )
```

- `NV_Ith_OMP`

  This macro gives access to the individual components of the data array of an `N_Vector`.

  The assignment `r = NV_Ith_OMP(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_OMP(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

  Here i ranges from 0 to $n-1$ for a vector of length n.

  Implementation:

  ```
  #define NV_Ith_OMP(v,i) ( NV_DATA_OMP(v)[i] )
  ```

The NVECTOR_OPENMP module defines OpenMP implementations of all vector operations listed in Table 6.2. Their names are obtained from those in Table 6.2 by appending the suffix `_OpenMP` (e.g. `N_VDestroy_OpenMP`). The module NVECTOR_OPENMP provides the following additional user-callable routines:

- `N_VNew_OpenMP`

  This function creates and allocates memory for a OpenMP `N_Vector`. Arguments are the vector length and number of threads.

  ```
  N_Vector N_VNew_OpenMP(sunindextype vec_length, int num_threads);
  ```

- `N_VNewEmpty_OpenMP`

  This function creates a new OpenMP `N_Vector` with an empty (`NULL`) data array.

  ```
  N_Vector N_VNewEmpty_OpenMP(sunindextype vec_length, int num_threads);
  ```

- `N_VMake_OpenMP`

  This function creates and allocates memory for a OpenMP vector with user-provided data array.

  (This function does *not* allocate memory for `v_data` itself.)

  ```
  N_Vector N_VMake_OpenMP(sunindextype vec_length, realtype *v_data, int num_threads);
  ```

- `N_VCloneVectorArray_OpenMP`

  This function creates (by cloning) an array of `count` OpenMP vectors.

  ```
  N_Vector *N_VCloneVectorArray_OpenMP(int count, N_Vector w);
  ```

- `N_VCloneVectorArrayEmpty_OpenMP`

  This function creates (by cloning) an array of `count` OpenMP vectors, each with an empty (`NULL`) data array.

  ```
  N_Vector *N_VCloneVectorArrayEmpty_OpenMP(int count, N_Vector w);
  ```

- `N_VDestroyVectorArray_OpenMP`

  This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_OpenMP` or with `N_VCloneVectorArrayEmpty_OpenMP`.

  ```
  void N_VDestroyVectorArray_OpenMP(N_Vector *vs, int count);
  ```

- `N_VGetLength_OpenMP`

  This function returns number of vector elements.

  ```
  sunindextype N_VGetLength_OpenMP(N_Vector v);
  ```

- N_VPrint_OpenMP

  This function prints the content of a OpenMP vector to stdout.

  ```
  void N_VPrint_OpenMP(N_Vector v);
  ```

Notes

- When looping over the components of an N_Vector v, it is more efficient to first obtain the component array via v_data = NV_DATA_OMP(v) and then access v_data[i] within the loop than it is to use NV_Ith_OMP(v,i) within the loop.

- N_VNewEmpty_OpenMP, N_VMake_OpenMP, and N_VCloneVectorArrayEmpty_OpenMP set the field *own_data* = SUNFALSE. N_VDestroy_OpenMP and N_VDestroyVectorArray_OpenMP will not attempt to free the pointer *data* for any N_Vector with *own_data* set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the *data* pointer.

- To maximize efficiency, vector operations in the NVECTOR_OPENMP implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

For solvers that include a Fortran interface module, the NVECTOR_OPENMP module also includes a Fortran-callable function FNVINITOMP(code, NEQ, NUMTHREADS, IER), to initialize this module. Here code is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); NEQ is the problem size (declared so as to match C type long int); NUMTHREADS is the number of threads; and IER is an error return flag equal 0 for success and -1 for failure.

6.4 The NVECTOR_PTHREADS implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR_OPENMP, and an implementation using Pthreads, called NVECTOR_PTHREADS. Testing has shown that vectors should be of length at least 100,000 before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The Pthreads NVECTOR implementation provided with SUNDIALS, denoted NVECTOR_PTHREADS, defines the *content* field of N_Vector to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag *own_data* which specifies the ownership of *data*, and the number of threads. Operations on the vector are threaded using POSIX threads (Pthreads).

```
struct _N_VectorContent_Pthreads {
  sunindextype length;
  booleantype own_data;
  realtype *data;
  int num_threads;
};
```

The header file to include when using this module is nvector_pthreads.h. The installed module library to link to is libsundials_nvecpthreads.*lib* where *.lib* is typically .so for shared libraries and .a for static libraries.

The following macros are provided to access the content of an NVECTOR_PTHREADS vector. The suffix _PT in the names denotes the Pthreads version.

- NV_CONTENT_PT

  This routine gives access to the contents of the Pthreads vector N_Vector.

The assignment `v_cont = NV_CONTENT_PT(v)` sets `v_cont` to be a pointer to the Pthreads `N_Vector` content structure.

Implementation:

```
#define NV_CONTENT_PT(v) ( (N_VectorContent_Pthreads)(v->content) )
```

- `NV_OWN_DATA_PT`, `NV_DATA_PT`, `NV_LENGTH_PT`, `NV_NUM_THREADS_PT`

  These macros give individual access to the parts of the content of a Pthreads `N_Vector`.

  The assignment `v_data = NV_DATA_PT(v)` sets `v_data` to be a pointer to the first component of the data for the `N_Vector` `v`. The assignment `NV_DATA_PT(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

  The assignment `v_len = NV_LENGTH_PT(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_PT(v) = len_v` sets the length of `v` to be `len_v`.

  The assignment `v_num_threads = NV_NUM_THREADS_PT(v)` sets `v_num_threads` to be the number of threads from `v`. On the other hand, the call `NV_NUM_THREADS_PT(v) = num_threads_v` sets the number of threads for `v` to be `num_threads_v`.

  Implementation:

  ```
  #define NV_OWN_DATA_PT(v) ( NV_CONTENT_PT(v)->own_data )
  #define NV_DATA_PT(v) ( NV_CONTENT_PT(v)->data )
  #define NV_LENGTH_PT(v) ( NV_CONTENT_PT(v)->length )
  #define NV_NUM_THREADS_PT(v) ( NV_CONTENT_PT(v)->num_threads )
  ```

- `NV_Ith_PT`

  This macro gives access to the individual components of the data array of an `N_Vector`.

  The assignment `r = NV_Ith_PT(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_PT(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

  Here i ranges from 0 to $n-1$ for a vector of length n.

  Implementation:

  ```
  #define NV_Ith_PT(v,i) ( NV_DATA_PT(v)[i] )
  ```

The NVECTOR_PTHREADS module defines Pthreads implementations of all vector operations listed in Table 6.2. Their names are obtained from those in Table 6.2 by appending the suffix `_Pthreads` (e.g. `N_VDestroy_Pthreads`). The module NVECTOR_PTHREADS provides the following additional user-callable routines:

- `N_VNew_Pthreads`

  This function creates and allocates memory for a Pthreads `N_Vector`. Arguments are the vector length and number of threads.

  ```
  N_Vector N_VNew_Pthreads(sunindextype vec_length, int num_threads);
  ```

- `N_VNewEmpty_Pthreads`

  This function creates a new Pthreads `N_Vector` with an empty (`NULL`) data array.

  ```
  N_Vector N_VNewEmpty_Pthreads(sunindextype vec_length, int num_threads);
  ```

- `N_VMake_Pthreads`

  This function creates and allocates memory for a Pthreads vector with user-provided data array.

  (This function does *not* allocate memory for `v_data` itself.)

  ```
  N_Vector N_VMake_Pthreads(sunindextype vec_length, realtype *v_data, int num_threads);
  ```

- N_VCloneVectorArray_Pthreads

  This function creates (by cloning) an array of count Pthreads vectors.

  ```
  N_Vector *N_VCloneVectorArray_Pthreads(int count, N_Vector w);
  ```

- N_VCloneVectorArrayEmpty_Pthreads

  This function creates (by cloning) an array of count Pthreads vectors, each with an empty (NULL) data array.

  ```
  N_Vector *N_VCloneVectorArrayEmpty_Pthreads(int count, N_Vector w);
  ```

- N_VDestroyVectorArray_Pthreads

  This function frees memory allocated for the array of count variables of type N_Vector created with N_VCloneVectorArray_Pthreads or with N_VCloneVectorArrayEmpty_Pthreads.

  ```
  void N_VDestroyVectorArray_Pthreads(N_Vector *vs, int count);
  ```

- N_VGetLength_Pthreads

  This function returns the number of vector elements.

  ```
  sunindextype N_VGetLength_Pthreads(N_Vector v);
  ```

- N_VPrint_Pthreads

  This function prints the content of a Pthreads vector to stdout.

  ```
  void N_VPrint_Pthreads(N_Vector v);
  ```

Notes

- When looping over the components of an N_Vector v, it is more efficient to first obtain the component array via v_data = NV_DATA_PT(v) and then access v_data[i] within the loop than it is to use NV_Ith_PT(v,i) within the loop.

- N_VNewEmpty_Pthreads, N_VMake_Pthreads, and N_VCloneVectorArrayEmpty_Pthreads set the field *own_data* = SUNFALSE. N_VDestroy_Pthreads and N_VDestroyVectorArray_Pthreads will not attempt to free the pointer *data* for any N_Vector with *own_data* set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the *data* pointer.

- To maximize efficiency, vector operations in the NVECTOR_PTHREADS implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

For solvers that include a Fortran interface module, the NVECTOR_PTHREADS module also includes a Fortran-callable function FNVINITPTS(code, NEQ, NUMTHREADS, IER), to initialize this module. Here code is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); NEQ is the problem size (declared so as to match C type long int); NUMTHREADS is the number of threads; and IER is an error return flag equal 0 for success and -1 for failure.

6.5 The NVECTOR_PARHYP implementation

The NVECTOR_PARHYP implementation of the NVECTOR module provided with SUNDIALS is a wrapper around *hypre*'s ParVector class. Most of the vector kernels simply call *hypre* vector operations. The implementation defines the *content* field of N_Vector to be a structure containing the global and local lengths of the vector, a pointer to an object of type hypre_ParVector, an MPI communicator, and a boolean flag *own_parvector* indicating ownership of the *hypre* parallel vector object x.

```
struct _N_VectorContent_ParHyp {
  sunindextype local_length;
  sunindextype global_length;
  booleantype own_parvector;
  MPI_Comm comm;
  hypre_ParVector *x;
};
```

The header file to include when using this module is `nvector_parhyp.h`. The installed module library to link to is `libsundials_nvecparhyp.`*`lib`* where *`.lib`* is typically `.so` for shared libraries and `.a` for static libraries.

Unlike native SUNDIALS vector types, NVECTOR_PARHYP does not provide macros to access its member variables. Note that NVECTOR_PARHYP requires SUNDIALS to be built with MPI support.

The NVECTOR_PARHYP module defines implementations of all vector operations listed in Table 6.2, except for `N_VSetArrayPointer` and `N_VGetArrayPointer`, because accessing raw vector data is handled by low-level *hypre* functions. As such, this vector is not available for use with SUNDIALS Fortran interfaces. When access to raw vector data is needed, one should extract the *hypre* vector first, and then use *hypre* methods to access the data. Usage examples of NVECTOR_PARHYP are provided in the `cvAdvDiff_non_ph.c` example program for CVODE [19] and the `ark_diurnal_kry_ph.c` example program for ARKODE [24].

The names of parhyp methods are obtained from those in Table 6.2 by appending the suffix `_ParHyp` (e.g. `N_VDestroy_ParHyp`). The module NVECTOR_PARHYP provides the following additional user-callable routines:

- `N_VNewEmpty_ParHyp`

  This function creates a new parhyp `N_Vector` with the pointer to the *hypre* vector set to `NULL`.

  ```
  N_Vector N_VNewEmpty_ParHyp(MPI_Comm comm,
                              sunindextype local_length,
                              sunindextype global_length);
  ```

- `N_VMake_ParHyp`

  This function creates an `N_Vector` wrapper around an existing *hypre* parallel vector. It does *not* allocate memory for `x` itself.

  ```
  N_Vector N_VMake_ParHyp(hypre_ParVector *x);
  ```

- `N_VGetVector_ParHyp`

  This function returns a pointer to the underlying *hypre* vector.

  ```
  hypre_ParVector *N_VGetVector_ParHyp(N_Vector v);
  ```

- `N_VCloneVectorArray_ParHyp`

  This function creates (by cloning) an array of `count` parallel vectors.

  ```
  N_Vector *N_VCloneVectorArray_ParHyp(int count, N_Vector w);
  ```

- `N_VCloneVectorArrayEmpty_ParHyp`

  This function creates (by cloning) an array of `count` parallel vectors, each with an empty (`NULL`) data array.

  ```
  N_Vector *N_VCloneVectorArrayEmpty_ParHyp(int count, N_Vector w);
  ```

- N_VDestroyVectorArray_ParHyp

  This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_ParHyp` or with `N_VCloneVectorArrayEmpty_ParHyp`.

  ```
  void N_VDestroyVectorArray_ParHyp(N_Vector *vs, int count);
  ```

- N_VPrint_ParHyp

  This function prints the content of a parhyp vector to stdout.

  ```
  void N_VPrint_ParHyp(N_Vector v);
  ```

Notes

- When there is a need to access components of an `N_Vector_ParHyp`, `v`, it is recommended to extract the *hypre* vector via `x_vec = N_VGetVector_ParHyp(v)` and then access components using appropriate *hypre* functions.

- `N_VNewEmpty_ParHyp`, `N_VMake_ParHyp`, and `N_VCloneVectorArrayEmpty_ParHyp` set the field *own_parvector* to `SUNFALSE`. `N_VDestroy_ParHyp` and `N_VDestroyVectorArray_ParHyp` will not attempt to delete an underlying *hypre* vector for any `N_Vector` with *own_parvector* set to `SUNFALSE`. In such a case, it is the user's responsibility to delete the underlying vector.

- To maximize efficiency, vector operations in the NVECTOR_PARHYP implementation that have more than one `N_Vector` argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

6.6 The NVECTOR_PETSC implementation

The NVECTOR_PETSC module is an NVECTOR wrapper around the PETSc vector. It defines the *content* field of a `N_Vector` to be a structure containing the global and local lengths of the vector, a pointer to the PETSc vector, an MPI communicator, and a boolean flag *own_data* indicating ownership of the wrapped PETSc vector.

```
struct _N_VectorContent_Petsc {
  sunindextype local_length;
  sunindextype global_length;
  booleantype own_data;
  Vec *pvec;
  MPI_Comm comm;
};
```

The header file to include when using this module is `nvector_petsc.h`. The installed module library to link to is `libsundials_nvecpetsc.`*`lib`* where *`.lib`* is typically `.so` for shared libraries and `.a` for static libraries.

Unlike native SUNDIALS vector types, NVECTOR_PETSC does not provide macros to access its member variables. Note that NVECTOR_PETSC requires SUNDIALS to be built with MPI support.

The NVECTOR_PETSC module defines implementations of all vector operations listed in Table 6.2, except for `N_VGetArrayPointer` and `N_VSetArrayPointer`. As such, this vector cannot be used with SUNDIALS Fortran interfaces. When access to raw vector data is needed, it is recommended to extract the PETSc vector first, and then use PETSc methods to access the data. Usage examples of NVECTOR_PETSC are provided in example programs for IDA [18].

The names of vector operations are obtained from those in Table 6.2 by appending the suffix `_Petsc` (e.g. `N_VDestroy_Petsc`). The module NVECTOR_PETSC provides the following additional user-callable routines:

- `N_VNewEmpty_Petsc`

  This function creates a new NVECTOR wrapper with the pointer to the wrapped PETSc vector set to (`NULL`). It is used by the `N_VMake_Petsc` and `N_VClone_Petsc` implementations.

  ```
  N_Vector N_VNewEmpty_Petsc(MPI_Comm comm,
                             sunindextype local_length,
                             sunindextype global_length);
  ```

- `N_VMake_Petsc`

  This function creates and allocates memory for an NVECTOR_PETSC wrapper around a user-provided PETSc vector. It does *not* allocate memory for the vector `pvec` itself.

  ```
  N_Vector N_VMake_Petsc(Vec *pvec);
  ```

- `N_VGetVector_Petsc`

  This function returns a pointer to the underlying PETSc vector.

  ```
  Vec *N_VGetVector_Petsc(N_Vector v);
  ```

- `N_VCloneVectorArray_Petsc`

  This function creates (by cloning) an array of `count` NVECTOR_PETSC vectors.

  ```
  N_Vector *N_VCloneVectorArray_Petsc(int count, N_Vector w);
  ```

- `N_VCloneVectorArrayEmpty_Petsc`

  This function creates (by cloning) an array of `count` NVECTOR_PETSC vectors, each with pointers to PETSc vectors set to (`NULL`).

  ```
  N_Vector *N_VCloneEmptyVectorArray_Petsc(int count, N_Vector w);
  ```

- `N_VDestroyVectorArray_Petsc`

  This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_Petsc` or with `N_VCloneVectorArrayEmpty_Petsc`.

  ```
  void N_VDestroyVectorArray_Petsc(N_Vector *vs, int count);
  ```

- `N_VPrint_Petsc`

  This function prints the content of a wrapped PETSc vector to stdout.

  ```
  void N_VPrint_Petsc(N_Vector v);
  ```

Notes

- When there is a need to access components of an `N_Vector_Petsc`, `v`, it is recommeded to extract the PETSc vector via `x_vec = N_VGetVector_Petsc(v)` and then access components using appropriate PETSc functions.

- The functions `N_VNewEmpty_Petsc`, `N_VMake_Petsc`, and `N_VCloneVectorArrayEmpty_Petsc` set the field *own_data* to `SUNFALSE`. `N_VDestroy_Petsc` and `N_VDestroyVectorArray_Petsc` will not attempt to free the pointer *pvec* for any `N_Vector` with *own_data* set to `SUNFALSE`. In such a case, it is the user's responsibility to deallocate the *pvec* pointer.

- To maximize efficiency, vector operations in the NVECTOR_PETSC implementation that have more than one `N_Vector` argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

6.7 The NVECTOR_CUDA implementation

The NVECTOR_CUDA module is an experimental NVECTOR implementation in the CUDA language. The module allows for SUNDIALS vector kernels to run on GPU devices. It is intended for users who are already familiar with CUDA and GPU programming. Building this vector module requires a CUDA compiler and, by extension, a C++ compiler. The class `Vector` in namespace `suncudavec` manages vector data layout:

```
template <class T, class I>
class Vector {
  I size_;
  I mem_size_;
  T* h_vec_;
  T* d_vec_;
  StreamPartitioning<T, I>* partStream_;
  ReducePartitioning<T, I>* partReduce_;
  bool ownPartitioning_;

  ...
};
```

The class members are vector size (length), size of the vector data memory block, pointers to vector data on the host and the device, pointers to classes `StreamPartitioning` and `ReducePartitioning`, which handle thread partitioning for streaming and reduction vector kernels, respectively, and a boolean flag that signals if the vector owns the thread partitioning. The class `Vector` inherits from the empty structure

```
struct _N_VectorContent_Cuda {
};
```

to interface the C++ class with the NVECTOR C code. When instantiated, the class `Vector` will allocate memory on both the host and the device. Due to the rapid progress of CUDA development, we expect that the `suncudavec::Vector` class will change frequently in future SUNDIALS releases. The code is structured so that it can tolerate significant changes in the `suncudavec::Vector` class without requiring changes to the user API.

The header file to include when using this module is `nvector_cuda.h`. The installed module library to link to is `libsundials_nveccuda.`*`lib`* where *`.lib`* is typically `.so` for shared libraries and `.a` for static libraries.

Unlike other native SUNDIALS vector types, NVECTOR_CUDA does not provide macros to access its member variables.

The NVECTOR_CUDA module defines implementations of all vector operations listed in Table 6.2, except for `N_VGetArrayPointer` and `N_VSetArrayPointer`. As such, this vector cannot be used with SUNDIALS Fortran interfaces, nor with SUNDIALS direct solvers and preconditioners. This support will be added in subsequent SUNDIALS releases. The NVECTOR_CUDA module provides separate functions to access data on the host and on the device. It also provides methods for copying from the host to the device and vice versa. Usage examples of NVECTOR_CUDA are provided in some example programs for CVODE [19].

The names of vector operations are obtained from those in Table 6.2 by appending the suffix `_Cuda` (e.g. `N_VDestroy_Cuda`). The module NVECTOR_CUDA provides the following additional user-callable routines:

- `N_VNew_Cuda`

  This function creates and allocates memory for a CUDA `N_Vector`. The memory is allocated on both host and device. Its only argument is the vector length.

  ```
  N_Vector N_VNew_Cuda(sunindextype vec_length);
  ```

- `N_VNewEmpty_Cuda`

  This function creates a new NVECTOR wrapper with the pointer to the wrapped CUDA vector set to (`NULL`). It is used by the `N_VNew_Cuda`, `N_VMake_Cuda`, and `N_VClone_Cuda` implementations.

  ```
  N_Vector N_VNewEmpty_Cuda(sunindextype vec_length);
  ```

- `N_VMake_Cuda`

  This function creates and allocates memory for an NVECTOR_CUDA wrapper around a user-provided `suncudavec::Vector` class. Its only argument is of type `N_VectorContent_Cuda`, which is the pointer to the class.

  ```
  N_Vector N_VMake_Cuda(N_VectorContent_Cuda c);
  ```

- `N_VCloneVectorArray_Cuda`

  This function creates (by cloning) an array of `count` NVECTOR_CUDA vectors.

  ```
  N_Vector *N_VCloneVectorArray_Cuda(int count, N_Vector w);
  ```

- `N_VCloneVectorArrayEmpty_Cuda`

  This function creates (by cloning) an array of `count` NVECTOR_CUDA vectors, each with pointers to CUDA vectors set to (`NULL`).

  ```
  N_Vector *N_VCloneEmptyVectorArray_Cuda(int count, N_Vector w);
  ```

- `N_VDestroyVectorArray_Cuda`

  This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_Cuda` or with `N_VCloneVectorArrayEmpty_Cuda`.

  ```
  void N_VDestroyVectorArray_Cuda(N_Vector *vs, int count);
  ```

- `N_VGetLength_Cuda`

  This function returns the length of the vector.

  ```
  sunindextype N_VGetLength_Cuda(N_Vector v);
  ```

- `N_VGetHostArrayPointer_Cuda`

  This function returns a pointer to the vector data on the host.

  ```
  realtype *N_VGetHostArrayPointer_Cuda(N_Vector v);
  ```

- `N_VGetDeviceArrayPointer_Cuda`

  This function returns a pointer to the vector data on the device.

  ```
  realtype *N_VGetDeviceArrayPointer_Cuda(N_Vector v);
  ```

- `N_VCopyToDevice_Cuda`

  This function copies host vector data to the device.

  ```
  realtype *N_VCopyToDevice_Cuda(N_Vector v);
  ```

- `N_VCopyFromDevice_Cuda`

  This function copies vector data from the device to the host.

  ```
  realtype *N_VCopyFromDevice_Cuda(N_Vector v);
  ```

- `N_VPrint_Cuda`

  This function prints the content of a wrapped CUDA vector to stdout.

  ```
  void N_VPrint_Cuda(N_Vector v);
  ```

Notes

- When there is a need to access components of an `N_Vector_Cuda`, `v`, it is recommeded to use functions `N_VGetDeviceArrayPointer_Cuda` or `N_VGetHostArrayPointer_Cuda`.
- To maximize efficiency, vector operations in the NVECTOR_CUDA implementation that have more than one `N_Vector` argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

6.8 The NVECTOR_RAJA implementation

The NVECTOR_RAJA module is an experimental NVECTOR implementation using the RAJA hardware abstraction layer, https://software.llnl.gov/RAJA/. In this implementation, RAJA allows for SUNDIALS vector kernels to run on GPU devices. The module is intended for users who are already familiar with RAJA and GPU programming. Building this vector module requires a C++11 compliant compiler and a CUDA software development toolkit. Besides the CUDA backend, RAJA has other backends such as serial, OpenMP, and OpenAC. These backends are not used in this SUNDIALS release. Class `Vector` in namespace `sunrajavec` manages the vector data layout:

```
template <class T, class I>
class Vector {
  I size_;
  I mem_size_;
  T* h_vec_;
  T* d_vec_;

  ...
};
```

The class members are: vector size (length), size of the vector data memory block, and pointers to vector data on the host and on the device. The class `Vector` inherits from an empty structure

```
struct _N_VectorContent_Raja {
};
```

to interface the C++ class with the NVECTOR C code. When instantiated, the class `Vector` will allocate memory on both the host and the device. Due to the rapid progress of RAJA development, we expect that the `sunrajavec::Vector` class will change frequently in future SUNDIALS releases. The code is structured so that it can tolerate significant changes in the `sunrajavec::Vector` class without requiring changes to the user API.

The header file to include when using this module is `nvector_raja.h`. The installed module library to link to is `libsundials_nvecraja.`*`lib`* where *`.lib`* is typically `.so` for shared libraries and `.a` for static libraries.

Unlike other native SUNDIALS vector types, NVECTOR_RAJA does not provide macros to access its member variables.

The NVECTOR_RAJA module defines the implementations of all vector operations listed in Table 6.2, except for `N_VGetArrayPointer` and `N_VSetArrayPointer`. As such, this vector cannot be used with SUNDIALS Fortran interfaces, nor with SUNDIALS direct solvers and preconditioners. The NVECTOR_RAJA module provides separate functions to access data on the host and on the device. It also provides methods for copying data from the host to the device and vice versa. Usage examples of NVECTOR_RAJA are provided in some example programs for CVODE [19].

The names of vector operations are obtained from those in Table 6.2 by appending the suffix `_Raja` (e.g. `N_VDestroy_Raja`). The module NVECTOR_RAJA provides the following additional user-callable routines:

- `N_VNew_Raja`

  This function creates and allocates memory for a RAJA `N_Vector`. The memory is allocated on both the host and the device. Its only argument is the vector length.

  ```
  N_Vector N_VNew_Raja(sunindextype vec_length);
  ```

- `N_VNewEmpty_Raja`

  This function creates a new NVECTOR wrapper with the pointer to the wrapped RAJA vector set to (`NULL`). It is used by the `N_VNew_Raja`, `N_VMake_Raja`, and `N_VClone_Raja` implementations.

  ```
  N_Vector N_VNewEmpty_Raja(sunindextype vec_length);
  ```

- `N_VMake_Raja`

  This function creates and allocates memory for an NVECTOR_RAJA wrapper around a user-provided `sunrajavec::Vector` class. Its only argument is of type `N_VectorContent_Raja`, which is the pointer to the class.

  ```
  N_Vector N_VMake_Raja(N_VectorContent_Raja c);
  ```

- `N_VCloneVectorArray_Raja`

  This function creates (by cloning) an array of `count` NVECTOR_RAJA vectors.

  ```
  N_Vector *N_VCloneVectorArray_Raja(int count, N_Vector w);
  ```

- `N_VCloneVectorArrayEmpty_Raja`

  This function creates (by cloning) an array of `count` NVECTOR_RAJA vectors, each with pointers to RAJA vectors set to (`NULL`).

  ```
  N_Vector *N_VCloneEmptyVectorArray_Raja(int count, N_Vector w);
  ```

- `N_VDestroyVectorArray_Raja`

  This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_Raja` or with `N_VCloneVectorArrayEmpty_Raja`.

  ```
  void N_VDestroyVectorArray_Raja(N_Vector *vs, int count);
  ```

- `N_VGetLength_Raja`

  This function returns the length of the vector.

  ```
  sunindextype N_VGetLength_Raja(N_Vector v);
  ```

- `N_VGetHostArrayPointer_Raja`

  This function returns a pointer to the vector data on the host.

  ```
  realtype *N_VGetHostArrayPointer_Raja(N_Vector v);
  ```

- `N_VGetDeviceArrayPointer_Raja`

  This function returns a pointer to the vector data on the device.

  ```
  realtype *N_VGetDeviceArrayPointer_Raja(N_Vector v);
  ```

- `N_VCopyToDevice_Raja`

  This function copies host vector data to the device.

  ```
  realtype *N_VCopyToDevice_Raja(N_Vector v);
  ```

- `N_VCopyFromDevice_Raja`

  This function copies vector data from the device to the host.

  ```
  realtype *N_VCopyFromDevice_Raja(N_Vector v);
  ```

- `N_VPrint_Raja`

  This function prints the content of a wrapped RAJA vector to stdout.

  ```
  void N_VPrint_Raja(N_Vector v);
  ```

Notes

- When there is a need to access components of an `N_Vector_Raja`, `v`, it is recommeded to use functions `N_VGetDeviceArrayPointer_Raja` or `N_VGetHostArrayPointer_Raja`.
- To maximize efficiency, vector operations in the NVECTOR_RAJA implementation that have more than one `N_Vector` argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

6.9 NVECTOR Examples

There are `NVector` examples that may be installed for the implementations provided with SUNDIALS. Each implementation makes use of the functions in `test_nvector.c`. These example functions show simple usage of the `NVector` family of functions. The input to the examples are the vector length, number of threads (if threaded implementation), and a print timing flag.
The following is a list of the example functions in `test_nvector.c`:

- `Test_N_VClone`: Creates clone of vector and checks validity of clone.
- `Test_N_VCloneEmpty`: Creates clone of empty vector and checks validity of clone.
- `Test_N_VCloneVectorArray`: Creates clone of vector array and checks validity of cloned array.
- `Test_N_VCloneVectorArray`: Creates clone of empty vector array and checks validity of cloned array.
- `Test_N_VGetArrayPointer`: Get array pointer.
- `Test_N_VSetArrayPointer`: Allocate new vector, set pointer to new vector array, and check values.
- `Test_N_VLinearSum` Case 1a: Test y = x + y
- `Test_N_VLinearSum` Case 1b: Test y = -x + y
- `Test_N_VLinearSum` Case 1c: Test y = ax + y
- `Test_N_VLinearSum` Case 2a: Test x = x + y
- `Test_N_VLinearSum` Case 2b: Test x = x - y
- `Test_N_VLinearSum` Case 2c: Test x = x + by
- `Test_N_VLinearSum` Case 3: Test z = x + y
- `Test_N_VLinearSum` Case 4a: Test z = x - y
- `Test_N_VLinearSum` Case 4b: Test z = -x + y
- `Test_N_VLinearSum` Case 5a: Test z = x + by

- `Test_N_VLinearSum` Case 5b: Test z = ax + y
- `Test_N_VLinearSum` Case 6a: Test z = -x + by
- `Test_N_VLinearSum` Case 6b: Test z = ax - y
- `Test_N_VLinearSum` Case 7: Test z = a(x + y)
- `Test_N_VLinearSum` Case 8: Test z = a(x - y)
- `Test_N_VLinearSum` Case 9: Test z = ax + by
- `Test_N_VConst`: Fill vector with constant and check result.
- `Test_N_VProd`: Test vector multiply: z = x * y
- `Test_N_VDiv`: Test vector division: z = x / y
- `Test_N_VScale`: Case 1: scale: x = cx
- `Test_N_VScale`: Case 2: copy: z = x
- `Test_N_VScale`: Case 3: negate: z = -x
- `Test_N_VScale`: Case 4: combination: z = cx
- `Test_N_VAbs`: Create absolute value of vector.
- `Test_N_VAddConst`: add constant vector: z = c + x
- `Test_N_VDotProd`: Calculate dot product of two vectors.
- `Test_N_VMaxNorm`: Create vector with known values, find and validate max norm.
- `Test_N_VWrmsNorm`: Create vector of known values, find and validate weighted root mean square.
- `Test_N_VWrmsNormMask`: Case 1: Create vector of known values, find and validate weighted root mean square using all elements.
- `Test_N_VWrmsNormMask`: Case 2: Create vector of known values, find and validate weighted root mean square using no elements.
- `Test_N_VMin`: Create vector, find and validate the min.
- `Test_N_VWL2Norm`: Create vector, find and validate the weighted Euclidean L2 norm.
- `Test_N_VL1Norm`: Create vector, find and validate the L1 norm.
- `Test_N_VCompare`: Compare vector with constant returning and validating comparison vector.
- `Test_N_VInvTest`: Test z[i] = 1 / x[i]
- `Test_N_VConstrMask`: Test mask of vector x with vector c.
- `Test_N_VMinQuotient`: Fill two vectors with known values. Calculate and validate minimum quotient.

6.10 NVECTOR functions used by KINSOL

In Table 6.3 below, we list the vector functions in the NVECTOR module used within the KINSOL package. The table also shows, for each function, which of the code modules uses the function. The KINSOL column shows function usage within the main solver module, while the remaining five columns show function usage within each of the KINSOL linear solver interfaces, the KINBBDPRE preconditioner module, and the FKINSOL module. Here KINDLS stands for the direct linear solver interface in KINSOL, and KINSPILS stands for the scaled, preconditioned, iterative linear solver interface in KINSOL.

At this point, we should emphasize that the KINSOL user does not need to know anything about the usage of vector functions by the KINSOL code modules in order to use KINSOL. The information is presented as an implementation detail for the interested reader.

Table 6.3: List of vector functions usage by KINSOL code modules

	KINSOL	KINDLS	KINSPILS	KINBBDPRE	FKINSOL
N_VGetVectorID					
N_VClone	✓		✓	✓	
N_VCloneEmpty					✓
N_VDestroy	✓		✓	✓	✓
N_VSpace	✓				
N_VGetArrayPointer		✓		✓	✓
N_VSetArrayPointer		✓			✓
N_VLinearSum	✓	✓	✓		
N_VConst			✓		
N_VProd	✓	✓	✓		
N_VDiv	✓		✓		
N_VScale	✓	✓	✓	✓	
N_VAbs	✓				
N_VInv	✓				
N_VDotProd	✓	✓	✓		
N_VMaxNorm	✓				
N_VMin	✓				
N_VWL2Norm	✓		✓		
N_VL1Norm			✓		
N_VConstrMask	✓				
N_VMinQuotient	✓				

The vector functions listed in Table 6.2 that are *not* used by KINSOL are `N_VAddConst`, `N_VWrmsNorm`, `N_VWrmsNormMask`, `N_VCompare`, and `N_VInvTest`. Therefore a user-supplied NVECTOR module for KINSOL could omit these functions.

Chapter 7

Description of the SUNMatrix module

For problems that involve direct methods for solving linear systems, the SUNDIALS solvers not only operate on generic vectors, but also on generic matrices (of type `SUNMatrix`), through a set of operations defined by the particular SUNMATRIX implementation. Users can provide their own specific implementation of the SUNMATRIX module, particularly in cases where they provide their own NVECTOR and/or linear solver modules, and require matrices that are compatible with those implementations. Alternately, we provide three SUNMATRIX implementations: dense, banded, and sparse. The generic operations are described below, and descriptions of the implementations provided with SUNDIALS follow.

The generic `SUNMatrix` type has been modeled after the object-oriented style of the generic `N_Vector` type. Specifically, a generic `SUNMatrix` is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the matrix, and an *ops* field pointing to a structure with generic matrix operations. The type `SUNMatrix` is defined as

```
typedef struct _generic_SUNMatrix *SUNMatrix;

struct _generic_SUNMatrix {
    void *content;
    struct _generic_SUNMatrix_Ops *ops;
};
```

The `_generic_SUNMatrix_Ops` structure is essentially a list of pointers to the various actual matrix operations, and is defined as

```
struct _generic_SUNMatrix_Ops {
  SUNMatrix_ID (*getid)(SUNMatrix);
  SUNMatrix    (*clone)(SUNMatrix);
  void         (*destroy)(SUNMatrix);
  int          (*zero)(SUNMatrix);
  int          (*copy)(SUNMatrix, SUNMatrix);
  int          (*scaleadd)(realtype, SUNMatrix, SUNMatrix);
  int          (*scaleaddi)(realtype, SUNMatrix);
  int          (*matvec)(SUNMatrix, N_Vector, N_Vector);
  int          (*space)(SUNMatrix, long int*, long int*);
};
```

The generic SUNMATRIX module defines and implements the matrix operations acting on `SUNMatrix` objects. These routines are nothing but wrappers for the matrix operations defined by a particular SUNMATRIX implementation, which are accessed through the *ops* field of the `SUNMatrix` structure. To

Table 7.1: Identifiers associated with matrix kernels supplied with SUNDIALS.

Matrix ID	Matrix type	ID Value
SUNMATRIX_DENSE	Dense M × N matrix	0
SUNMATRIX_BAND	Band M × M matrix	1
SUNMATRIX_SPARSE	Sparse (CSR or CSC) M × N matrix	2
SUNMATRIX_CUSTOM	User-provided custom matrix	3

illustrate this point we show below the implementation of a typical matrix operation from the generic SUNMATRIX module, namely `SUNMatZero`, which sets all values of a matrix `A` to zero, returning a flag denoting a successful/failed operation:

```
int SUNMatZero(SUNMatrix A)
{
  return((int) A->ops->zero(A));
}
```

Table 7.2 contains a complete list of all matrix operations defined by the generic SUNMATRIX module. A particular implementation of the SUNMATRIX module must:

- Specify the *content* field of the `SUNMatrix` object.
- Define and implement a minimal subset of the matrix operations. See the documentation for each SUNDIALS solver to determine which SUNMATRIX operations they require.

  Note that the names of these routines should be unique to that implementation in order to permit using more than one SUNMATRIX module (each with different `SUNMatrix` internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free a `SUNMatrix` with the new *content* field and with *ops* pointing to the new matrix operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined `SUNMatrix` (e.g., a routine to print the content for debugging purposes).
- Optionally, provide accessor macros or functions as needed for that particular implementation to access different parts of the *content* field of the newly defined `SUNMatrix`.

Each SUNMATRIX implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 7.1. It is recommended that a user-supplied SUNMATRIX implementation use the `SUNMATRIX_CUSTOM` identifier.

Table 7.2: Description of the `SUNMatrix` operations

Name	Usage and Description
SUNMatGetID	`id = SUNMatGetID(A);` Returns the type identifier for the matrix `A`. It is used to determine the matrix implementation type (e.g. dense, banded, sparse,...) from the abstract `SUNMatrix` interface. This is used to assess compatibility with SUNDIALS-provided linear solver implementations. Returned values are given in the Table 7.1.

continued on next page

Name	Usage and Description
SUNMatClone	`B = SUNMatClone(A);` Creates a new `SUNMatrix` of the same type as an existing matrix `A` and sets the *ops* field. It does not copy the matrix, but rather allocates storage for the new matrix.
SUNMatDestroy	`SUNMatDestroy(A);` Destroys the `SUNMatrix` `A` and frees memory allocated for its internal data.
SUNMatSpace	`ier = SUNMatSpace(A, &lrw, &liw);` Returns the storage requirements for the matrix `A`. `lrw` is a `long int` containing the number of realtype words and `liw` is a `long int` containing the number of integer words. The return value is an integer flag denoting success/failure of the operation. This function is advisory only, for use in determining a user's total space requirements; it could be a dummy function in a user-supplied SUNMATRIX module if that information is not of interest.
SUNMatZero	`ier = SUNMatZero(A);` Performs the operation $A_{ij} = 0$ for all entries of the matrix A. The return value is an integer flag denoting success/failure of the operation.
SUNMatCopy	`ier = SUNMatCopy(A,B);` Performs the operation $B_{ij} = A_{i,j}$ for all entries of the matrices A and B. The return value is an integer flag denoting success/failure of the operation.
SUNMatScaleAdd	`ier = SUNMatScaleAdd(c, A, B);` Performs the operation $A = cA + B$. The return value is an integer flag denoting success/failure of the operation.
SUNMatScaleAddI	`ier = SUNMatScaleAddI(c, A);` Performs the operation $A = cA + I$. The return value is an integer flag denoting success/failure of the operation.
SUNMatMatvec	`ier = SUNMatMatvec(A, x, y);` Performs the matrix-vector product operation, $y = Ax$. It should only be called with vectors `x` and `y` that are compatible with the matrix `A` – both in storage type and dimensions. The return value is an integer flag denoting success/failure of the operation.

We note that not all SUNMATRIX types are compatible with all NVECTOR types provided with SUNDIALS. This is primarily due to the need for compatibility within the `SUNMatMatvec` routine; however, compatibility between SUNMATRIX and NVECTOR implementations is more crucial when considering their interaction within SUNLINSOL objects, as will be described in more detail in Chapter 8. More specifically, in Table 7.3 we show the matrix interfaces available as SUNMATRIX modules, and the compatible vector implementations.

Table 7.3: SUNDIALS matrix interfaces and vector implementations that can be used for each.

Matrix Interface	Serial	Parallel (MPI)	OpenMP	pThreads	*hypre* Vec.	PETSc Vec.	CUDA	RAJA	User Suppl.
Dense	✓		✓	✓					✓

continued on next page

Matrix Interface	Serial	Parallel (MPI)	OpenMP	pThreads	*hypre* Vec.	PETSc Vec.	CUDA	RAJA	User Suppl.
Band	✓		✓	✓					✓
Sparse	✓		✓	✓					✓
User supplied	✓	✓	✓	✓	✓	✓	✓	✓	✓

7.1 The SUNMatrix_Dense implementation

The dense implementation of the SUNMATRIX module provided with SUNDIALS, SUNMATRIX_DENSE, defines the *content* field of `SUNMatrix` to be the following structure:

```
struct _SUNMatrixContent_Dense {
  sunindextype M;
  sunindextype N;
  realtype *data;
  sunindextype ldata;
  realtype **cols;
};
```

These entries of the *content* field contain the following information:

M - number of rows

N - number of columns

data - pointer to a contiguous block of `realtype` variables. The elements of the dense matrix are stored columnwise, i.e. the (`i`,`j`)-th element of a dense SUNMATRIX `A` (with $0 \leq$ `i` $<$ `M` and $0 \leq$ `j` $<$ `N`) may be accessed via `data[j*M+i]`.

ldata - length of the data array (= `M`·`N`).

cols - array of pointers. `cols[j]` points to the first element of the j-th column of the matrix in the array `data`. The (`i`,`j`)-th element of a dense SUNMATRIX `A` (with $0 \leq$ `i` $<$ `M` and $0 \leq$ `j` $<$ `N`) may be accessed via `cols[j][i]`.

The header file to include when using this module is `sunmatrix/sunmatrix_dense.h`. The SUNMATRIX_DENSE module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunmatrixdense` module library.

The following macros are provided to access the content of a SUNMATRIX_DENSE matrix. The prefix `SM_` in the names denotes that these macros are for *SUNMatrix* implementations, and the suffix `_D` denotes that these are specific to the *dense* version.

- `SM_CONTENT_D`

  This macro gives access to the contents of the dense `SUNMatrix`.

  The assignment `A_cont` = `SM_CONTENT_D(A)` sets `A_cont` to be a pointer to the dense `SUNMatrix` content structure.

  Implementation:

  ```
  #define SM_CONTENT_D(A)     ( (SUNMatrixContent_Dense)(A->content) )
  ```

- `SM_ROWS_D`, `SM_COLUMNS_D`, and `SM_LDATA_D`

  These macros give individual access to various lengths relevant to the content of a dense `SUNMatrix`.

These may be used either to retrieve or to set these values. For example, the assignment `A_rows = SM_ROWS_D(A)` sets `A_rows` to be the number of rows in the matrix `A`. Similarly, the assignment `SM_COLUMNS_D(A) = A_cols` sets the number of columns in `A` to equal `A_cols`.

Implementation:

```
#define SM_ROWS_D(A)        ( SM_CONTENT_D(A)->M )
#define SM_COLUMNS_D(A)     ( SM_CONTENT_D(A)->N )
#define SM_LDATA_D(A)       ( SM_CONTENT_D(A)->ldata )
```

- `SM_DATA_D` and `SM_COLS_D`

  These macros give access to the `data` and `cols` pointers for the matrix entries.

  The assignment `A_data = SM_DATA_D(A)` sets `A_data` to be a pointer to the first component of the data array for the dense `SUNMatrix` `A`. The assignment `SM_DATA_D(A) = A_data` sets the data array of `A` to be `A_data` by storing the pointer `A_data`.

  Similarly, the assignment `A_cols = SM_COLS_D(A)` sets `A_cols` to be a pointer to the array of column pointers for the dense `SUNMatrix` `A`. The assignment `SM_COLS_D(A) = A_cols` sets the column pointer array of `A` to be `A_cols` by storing the pointer `A_cols`.

  Implementation:

  ```
  #define SM_DATA_D(A)        ( SM_CONTENT_D(A)->data )
  #define SM_COLS_D(A)        ( SM_CONTENT_D(A)->cols )
  ```

- `SM_COLUMN_D` and `SM_ELEMENT_D`

  These macros give access to the individual columns and entries of the data array of a dense `SUNMatrix`.

  The assignment `col_j = SM_COLUMN_D(A,j)` sets `col_j` to be a pointer to the first entry of the `j`-th column of the $\mathtt{M} \times \mathtt{N}$ dense matrix `A` (with $0 \le \mathtt{j} < \mathtt{N}$). The type of the expression `SM_COLUMN_D(A,j)` is `realtype *`. The pointer returned by the call `SM_COLUMN_D(A,j)` can be treated as an array which is indexed from 0 to $\mathtt{M} - 1$.

  The assignments `SM_ELEMENT_D(A,i,j) = a_ij` and `a_ij = SM_ELEMENT_D(A,i,j)` reference the (`i`,`j`)-th element of the $\mathtt{M} \times \mathtt{N}$ dense matrix `A` (with $0 \le \mathtt{i} < \mathtt{M}$ and $0 \le \mathtt{j} < \mathtt{N}$).

  Implementation:

  ```
  #define SM_COLUMN_D(A,j)    ( (SM_CONTENT_D(A)->cols)[j] )
  #define SM_ELEMENT_D(A,i,j) ( (SM_CONTENT_D(A)->cols)[j][i] )
  ```

The SUNMATRIX_DENSE module defines dense implementations of all matrix operations listed in Table 7.2. Their names are obtained from those in Table 7.2 by appending the suffix `_Dense` (e.g. `SUNMatCopy_Dense`). The module SUNMATRIX_DENSE provides the following additional user-callable routines:

- `SUNDenseMatrix`

  This constructor function creates and allocates memory for a dense `SUNMatrix`. Its arguments are the number of rows, `M`, and columns, `N`, for the dense matrix.

  ```
  SUNMatrix SUNDenseMatrix(sunindextype M, sunindextype N);
  ```

- `SUNDenseMatrix_Print`

  This function prints the content of a dense `SUNMatrix` to the output stream specified by `outfile`. Note: `stdout` or `stderr` may be used as arguments for `outfile` to print directly to standard output or standard error, respectively.

  ```
  void SUNDenseMatrix_Print(SUNMatrix A, FILE* outfile);
  ```

- SUNDenseMatrix_Rows

  This function returns the number of rows in the dense SUNMatrix.

  ```
  sunindextype SUNDenseMatrix_Rows(SUNMatrix A);
  ```

- SUNDenseMatrix_Columns

  This function returns the number of columns in the dense SUNMatrix.

  ```
  sunindextype SUNDenseMatrix_Columns(SUNMatrix A);
  ```

- SUNDenseMatrix_LData

  This function returns the length of the data array for the dense SUNMatrix.

  ```
  sunindextype SUNDenseMatrix_LData(SUNMatrix A);
  ```

- SUNDenseMatrix_Data

  This function returns a pointer to the data array for the dense SUNMatrix.

  ```
  realtype* SUNDenseMatrix_Data(SUNMatrix A);
  ```

- SUNDenseMatrix_Cols

  This function returns a pointer to the cols array for the dense SUNMatrix.

  ```
  realtype** SUNDenseMatrix_Cols(SUNMatrix A);
  ```

- SUNDenseMatrix_Column

  This function returns a pointer to the first entry of the jth column of the dense SUNMatrix. The resulting pointer should be indexed over the range 0 to $\texttt{M} - 1$.

  ```
  realtype* SUNDenseMatrix_Column(SUNMatrix A, sunindextype j);
  ```

Notes

- When looping over the components of a dense SUNMatrix A, the most efficient approaches are to:
  - First obtain the component array via `A_data = SM_DATA_D(A)` or `A_data = SUNDenseMatrix_Data(A)` and then access `A_data[i]` within the loop.
  - First obtain the array of column pointers via `A_cols = SM_COLS_D(A)` or `A_cols = SUNDenseMatrix_Cols(A)`, and then access `A_cols[j][i]` within the loop.
  - Within a loop over the columns, access the column pointer via `A_colj = SUNDenseMatrix_Column(A,j)` and then to access the entries within that column using `A_colj[i]` within the loop.

  All three of these are more efficient than using `SM_ELEMENT_D(A,i,j)` within a double loop.

- Within the SUNMatMatvec_Dense routine, internal consistency checks are performed to ensure that the matrix is called with consistent NVECTOR implementations. These are currently limited to: NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS. As additional compatible vector implementations are added to SUNDIALS, these will be included within this compatibility check.

For solvers that include a Fortran interface module, the SUNMATRIX_DENSE module also includes the Fortran-callable function `FSUNDenseMatInit(code, M, N, ier)` to initialize this SUNMATRIX_DENSE module for a given SUNDIALS solver. Here `code` is an integer input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); `M` and `N` are the corresponding dense matrix construction arguments (declared to match C type `long int`); and `ier` is an error return flag equal to 0 for success and -1 for failure. Both `code` and `ier` are declared to match C type `int`. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function `FSUNDenseMassMatInit(M, N, ier)` initializes this SUNMATRIX_DENSE module for storing the mass matrix.

7.2 The SUNMatrix_Band implementation

The banded implementation of the SUNMATRIX module provided with SUNDIALS, SUNMATRIX_BAND, defines the *content* field of `SUNMatrix` to be the following structure:

```
struct _SUNMatrixContent_Band {
  sunindextype M;
  sunindextype N;
  sunindextype mu;
  sunindextype ml;
  sunindextype s_mu;
  sunindextype ldim;
  realtype *data;
  sunindextype ldata;
  realtype **cols;
};
```

A diagram of the underlying data representation in a banded matrix is shown in Figure 7.1. A more complete description of the parts of this *content* field is given below:

M - number of rows

N - number of columns (`N` = `M`)

mu - upper half-bandwidth, 0 ≤ `mu` < `N`

ml - lower half-bandwidth, 0 ≤ `ml` < `N`

s_mu - storage upper bandwidth, `mu` ≤ `s_mu` < `N`. The LU decomposition routines in the associated SUNLINSOL_BAND and SUNLINSOL_LAPACKBAND modules write the LU factors into the storage for A. The upper triangular factor U, however, may have an upper bandwidth as big as min(`N`-1,`mu`+`ml`) because of partial pivoting. The `s_mu` field holds the upper half-bandwidth allocated for A.

ldim - leading dimension (`ldim` ≥ `s_mu`)

data - pointer to a contiguous block of `realtype` variables. The elements of the banded matrix are stored columnwise (i.e. columns are stored one on top of the other in memory). Only elements within the specified half-bandwidths are stored. `data` is a pointer to `ldata` contiguous locations which hold the elements within the band of A.

ldata - length of the data array (= `ldim`·(`s_mu`+`ml`+1))

cols - array of pointers. `cols[j]` is a pointer to the uppermost element within the band in the j-th column. This pointer may be treated as an array indexed from `s_mu`−`mu` (to access the uppermost element within the band in the j-th column) to `s_mu`+`ml` (to access the lowest element within the band in the j-th column). Indices from 0 to `s_mu`−`mu`−1 give access to extra storage elements required by the LU decomposition function. Finally, `cols[j][i-j+s_mu]` is the (i, j)-th element with j−`mu` ≤ i ≤ j+`ml`.

The header file to include when using this module is `sunmatrix/sunmatrix_band.h`. The SUNMATRIX_BAND module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunmatrixband` module library.

The following macros are provided to access the content of a SUNMATRIX_BAND matrix. The prefix `SM_` in the names denotes that these macros are for *SUNMatrix* implementations, and the suffix `_B` denotes that these are specific to the *banded* version.

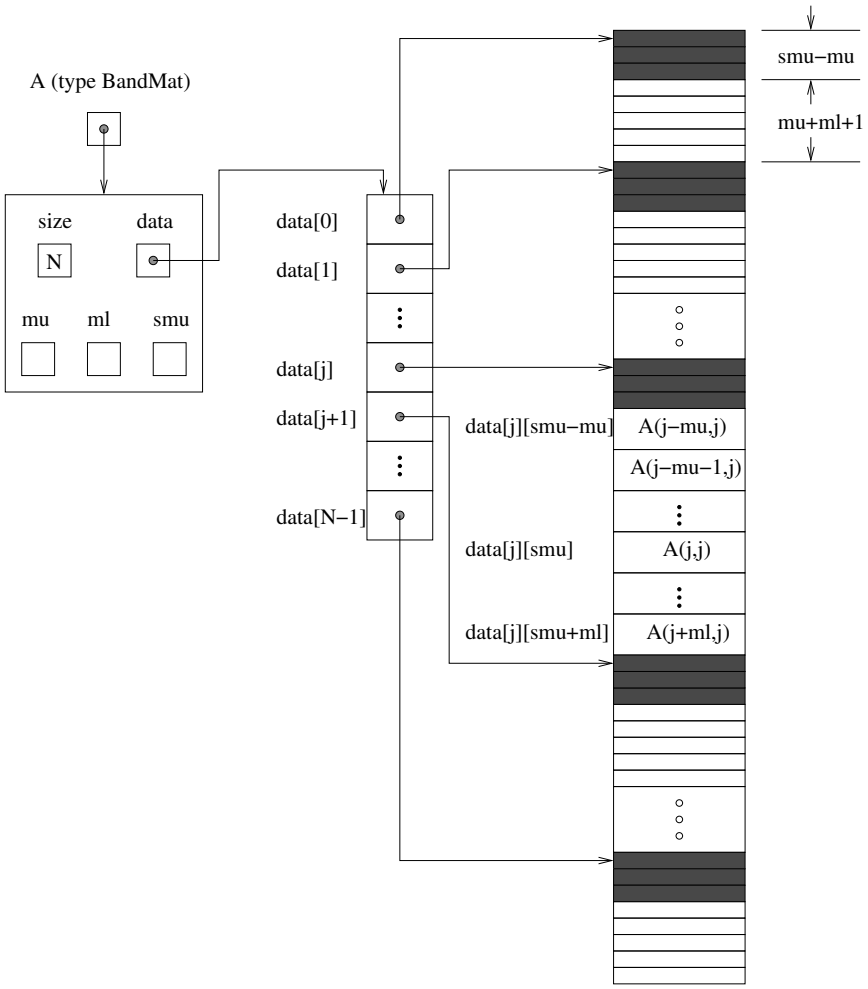


Figure 7.1: Diagram of the storage for the SUNMATRIX_BAND module. Here `A` is an $\texttt{N} \times \texttt{N}$ band matrix with upper and lower half-bandwidths `mu` and `ml`, respectively. The rows and columns of `A` are numbered from 0 to $\texttt{N} - 1$ and the (i, j)-th element of `A` is denoted `A(i,j)`. The greyed out areas of the underlying component storage are used by the associated SUNLINSOL_BAND linear solver.

- SM_CONTENT_B

  This routine gives access to the contents of the banded SUNMatrix.

  The assignment A_cont = SM_CONTENT_B(A) sets A_cont to be a pointer to the banded SUNMatrix content structure.

  Implementation:

  ```
  #define SM_CONTENT_B(A)     ( (SUNMatrixContent_Band)(A->content) )
  ```

- SM_ROWS_B, SM_COLUMNS_B, SM_UBAND_B, SM_LBAND_B, SM_SUBAND_B, SM_LDIM_B, and SM_LDATA_B

  These macros give individual access to various lengths relevant to the content of a banded SUNMatrix.

  These may be used either to retrieve or to set these values. For example, the assignment A_rows = SM_ROWS_B(A) sets A_rows to be the number of rows in the matrix A. Similarly, the assignment SM_COLUMNS_B(A) = A_cols sets the number of columns in A to equal A_cols.

  Implementation:

  ```
  #define SM_ROWS_B(A)        ( SM_CONTENT_B(A)->M )
  #define SM_COLUMNS_B(A)     ( SM_CONTENT_B(A)->N )
  #define SM_UBAND_B(A)       ( SM_CONTENT_B(A)->mu )
  #define SM_LBAND_B(A)       ( SM_CONTENT_B(A)->ml )
  #define SM_SUBAND_B(A)      ( SM_CONTENT_B(A)->s_mu )
  #define SM_LDIM_B(A)        ( SM_CONTENT_B(A)->ldim )
  #define SM_LDATA_B(A)       ( SM_CONTENT_B(A)->ldata )
  ```

- SM_DATA_B and SM_COLS_B

  These macros give access to the data and cols pointers for the matrix entries.

  The assignment A_data = SM_DATA_B(A) sets A_data to be a pointer to the first component of the data array for the banded SUNMatrix A. The assignment SM_DATA_B(A) = A_data sets the data array of A to be A_data by storing the pointer A_data.

  Similarly, the assignment A_cols = SM_COLS_B(A) sets A_cols to be a pointer to the array of column pointers for the banded SUNMatrix A. The assignment SM_COLS_B(A) = A_cols sets the column pointer array of A to be A_cols by storing the pointer A_cols.

  Implementation:

  ```
  #define SM_DATA_B(A)        ( SM_CONTENT_B(A)->data )
  #define SM_COLS_B(A)        ( SM_CONTENT_B(A)->cols )
  ```

- SM_COLUMN_B, SM_COLUMN_ELEMENT_B, and SM_ELEMENT_B

  These macros give access to the individual columns and entries of the data array of a banded SUNMatrix.

  The assignments SM_ELEMENT_B(A,i,j) = a_ij and a_ij = SM_ELEMENT_B(A,i,j) reference the (i,j)-th element of the N × N band matrix A, where $0 \le$ i, j $\le$ N − 1. The location (i,j) should further satisfy j−mu $\le$ i $\le$ j+ml.

  The assignment col_j = SM_COLUMN_B(A,j) sets col_j to be a pointer to the diagonal element of the j-th column of the N × N band matrix A, $0 \le$ j $\le$ N − 1. The type of the expression SM_COLUMN_B(A,j) is realtype *. The pointer returned by the call SM_COLUMN_B(A,j) can be treated as an array which is indexed from −mu to ml.

  The assignments SM_COLUMN_ELEMENT_B(col_j,i,j) = a_ij and a_ij = SM_COLUMN_ELEMENT_B(col_j,i,j) reference the (i,j)-th entry of the band matrix A when used in conjunction with SM_COLUMN_B to reference the j-th column through col_j. The index (i,j) should satisfy j−mu $\le$ i $\le$ j+ml.

Implementation:

```
#define SM_COLUMN_B(A,j)    ( ((SM_CONTENT_B(A)->cols)[j])+SM_SUBAND_B(A) )
#define SM_COLUMN_ELEMENT_B(col_j,i,j) (col_j[(i)-(j)])
#define SM_ELEMENT_B(A,i,j)
              ( (SM_CONTENT_B(A)->cols)[j][(i)-(j)+SM_SUBAND_B(A)] )
```

The SUNMATRIX_BAND module defines banded implementations of all matrix operations listed in Table 7.2. Their names are obtained from those in Table 7.2 by appending the suffix `_Band` (e.g. `SUNMatCopy_Band`). The module SUNMATRIX_BAND provides the following additional user-callable routines:

- `SUNBandMatrix`

  This constructor function creates and allocates memory for a banded `SUNMatrix`. Its arguments are the matrix size, `N`, the upper and lower half-bandwidths of the matrix, `mu` and `ml`, and the stored upper bandwidth, `smu`. When creating a band `SUNMatrix`, if the matrix will be used by the SUNLINSOL_BAND module then `smu` should be at least min(`N`-1,`mu`+`ml`); otherwise `smu` should be at least `mu`.

  ```
  SUNMatrix SUNBandMatrix(sunindextype N, sunindextype mu,
                          sunindextype ml, sunindextype smu);
  ```

- `SUNBandMatrix_Print`

  This function prints the content of a banded `SUNMatrix` to the output stream specified by `outfile`. Note: `stdout` or `stderr` may be used as arguments for `outfile` to print directly to standard output or standard error, respectively.

  ```
  void SUNBandMatrix_Print(SUNMatrix A, FILE* outfile);
  ```

- `SUNBandMatrix_Rows`

  This function returns the number of rows in the banded `SUNMatrix`.

  ```
  sunindextype SUNBandMatrix_Rows(SUNMatrix A);
  ```

- `SUNBandMatrix_Columns`

  This function returns the number of columns in the banded `SUNMatrix`.

  ```
  sunindextype SUNBandMatrix_Columns(SUNMatrix A);
  ```

- `SUNBandMatrix_LowerBandwidth`

  This function returns the lower half-bandwidth of the banded `SUNMatrix`.

  ```
  sunindextype SUNBandMatrix_LowerBandwidth(SUNMatrix A);
  ```

- `SUNBandMatrix_UpperBandwidth`

  This function returns the upper half-bandwidth of the banded `SUNMatrix`.

  ```
  sunindextype SUNBandMatrix_UpperBandwidth(SUNMatrix A);
  ```

- `SUNBandMatrix_StoredUpperBandwidth`

  This function returns the stored upper half-bandwidth of the banded `SUNMatrix`.

  ```
  sunindextype SUNBandMatrix_StoredUpperBandwidth(SUNMatrix A);
  ```

- `SUNBandMatrix_LDim`

  This function returns the length of the leading dimension of the banded `SUNMatrix`.

  ```
  sunindextype SUNBandMatrix_LDim(SUNMatrix A);
  ```

- SUNBandMatrix_Data

  This function returns a pointer to the data array for the banded SUNMatrix.

  ```
  realtype* SUNBandMatrix_Data(SUNMatrix A);
  ```

- SUNBandMatrix_Cols

  This function returns a pointer to the cols array for the banded SUNMatrix.

  ```
  realtype** SUNBandMatrix_Cols(SUNMatrix A);
  ```

- SUNBandMatrix_Column

  This function returns a pointer to the diagonal entry of the j-th column of the banded SUNMatrix. The resulting pointer should be indexed over the range $-$mu to ml.

  ```
  realtype* SUNBandMatrix_Column(SUNMatrix A, sunindextype j);
  ```

Notes

- When looping over the components of a banded SUNMatrix A, the most efficient approaches are to:
  - First obtain the component array via A_data = SM_DATA_B(A) or A_data = SUNBandMatrix_Data(A) and then access A_data[i] within the loop.
  - First obtain the array of column pointers via A_cols = SM_COLS_B(A) or A_cols = SUNBandMatrix_Cols(A), and then access A_cols[j][i] within the loop.
  - Within a loop over the columns, access the column pointer via A_colj = SUNBandMatrix_Column(A,j) and then to access the entries within that column using SM_COLUMN_ELEMENT_B(A_colj,i,j).

  All three of these are more efficient than using SM_ELEMENT_B(A,i,j) within a double loop.

- Within the SUNMatMatvec_Band routine, internal consistency checks are performed to ensure that the matrix is called with consistent NVECTOR implementations. These are currently limited to: NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS. As additional compatible vector implementations are added to SUNDIALS, these will be included within this compatibility check.

For solvers that include a Fortran interface module, the SUNMATRIX_BAND module also includes the Fortran-callable function FSUNBandMatInit(code, N, mu, ml, smu, ier) to initialize this SUNMATRIX_BAND module for a given SUNDIALS solver. Here code is an integer input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); N, mu, ml and smu are the corresponding band matrix construction arguments (declared to match C type long int); and ier is an error return flag equal to 0 for success and -1 for failure. Both code and ier are declared to match C type int. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function FSUNBandMassMatInit(N, mu, ml, smu, ier) initializes this SUNMATRIX_BAND module for storing the mass matrix.

7.3 The SUNMatrix_Sparse implementation

The sparse implementation of the SUNMATRIX module provided with SUNDIALS, SUNMATRIX_SPARSE, is designed to work with either *compressed-sparse-column* (CSC) or *compressed-sparse-row* (CSR) sparse matrix formats. To this end, it defines the *content* field of SUNMatrix to be the following structure:

```
struct _SUNMatrixContent_Sparse {
  sunindextype M;
  sunindextype N;
  sunindextype NNZ;
```

```
  sunindextype NP;
  realtype *data;
  int sparsetype;
  sunindextype *indexvals;
  sunindextype *indexptrs;
  /* CSC indices */
  sunindextype **rowvals;
  sunindextype **colptrs;
  /* CSR indices */
  sunindextype **colvals;
  sunindextype **rowptrs;
};
```

A diagram of the underlying data representation for a CSC matrix is shown in Figure 7.2 (the CSR format is similar). A more complete description of the parts of this *content* field is given below:

M - number of rows

N - number of columns

NNZ - maximum number of nonzero entries in the matrix (allocated length of `data` and `indexvals` arrays)

NP - number of index pointers (e.g. number of column pointers for CSC matrix). For CSC matrices `NP` = `N`, and for CSR matrices `NP` = `M`. This value is set automatically based the input for `sparsetype`.

data - pointer to a contiguous block of `realtype` variables (of length `NNZ`), containing the values of the nonzero entries in the matrix

sparsetype - type of the sparse matrix (`CSC_MAT` or `CSR_MAT`)

indexvals - pointer to a contiguous block of `int` variables (of length `NNZ`), containing the row indices (if CSC) or column indices (if CSR) of each nonzero matrix entry held in `data`

indexptrs - pointer to a contiguous block of `int` variables (of length `NP+1`). For CSC matrices each entry provides the index of the first column entry into the `data` and `indexvals` arrays, e.g. if `indexptr[3]=7`, then the first nonzero entry in the fourth column of the matrix is located in `data[7]`, and is located in row `indexvals[7]` of the matrix. The last entry contains the total number of nonzero values in the matrix and hence points one past the end of the active data in the `data` and `indexvals` arrays. For CSR matrices, each entry provides the index of the first row entry into the `data` and `indexvals` arrays.

The following pointers are added to the `SlsMat` type for user convenience, to provide a more intuitive interface to the CSC and CSR sparse matrix data structures. They are set automatically when creating a sparse SUNMATRIX, based on the sparse matrix storage type.

rowvals - pointer to `indexvals` when `sparsetype` is `CSC_MAT`, otherwise set to `NULL`.

colptrs - pointer to `indexptrs` when `sparsetype` is `CSC_MAT`, otherwise set to `NULL`.

colvals - pointer to `indexvals` when `sparsetype` is `CSR_MAT`, otherwise set to `NULL`.

rowptrs - pointer to `indexptrs` when `sparsetype` is `CSR_MAT`, otherwise set to `NULL`.

For example, the 5×4 CSC matrix

$$\begin{bmatrix} 0 & 3 & 1 & 0 \\ 3 & 0 & 0 & 2 \\ 0 & 7 & 0 & 0 \\ 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 5 \end{bmatrix}$$

could be stored in this structure as either

```
M = 5;
N = 4;
NNZ = 8;
NP = N;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0};
sparsetype = CSC_MAT;
indexvals = {1, 3, 0, 2, 0, 1, 3, 4};
indexptrs = {0, 2, 4, 5, 8};
```

or

```
M = 5;
N = 4;
NNZ = 10;
NP = N;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0, *, *};
sparsetype = CSC_MAT;
indexvals = {1, 3, 0, 2, 0, 1, 3, 4, *, *};
indexptrs = {0, 2, 4, 5, 8};
```

where the first has no unused space, and the second has additional storage (the entries marked with * may contain any values). Note in both cases that the final value in `indexptrs` is 8, indicating the total number of nonzero entries in the matrix.

Similarly, in CSR format, the same matrix could be stored as

```
M = 5;
N = 4;
NNZ = 8;
NP = N;
data = {3.0, 1.0, 3.0, 2.0, 7.0, 1.0, 9.0, 5.0};
sparsetype = CSR_MAT;
indexvals = {1, 2, 0, 3, 1, 0, 3, 3};
indexptrs = {0, 2, 4, 5, 7, 8};
```

The header file to include when using this module is `sunmatrix/sunmatrix_sparse.h`. The SUNMATRIX_SPARSE module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunmatrixsparse` module library.

The following macros are provided to access the content of a SUNMATRIX_SPARSE matrix. The prefix `SM_` in the names denotes that these macros are for *SUNMatrix* implementations, and the suffix `_S` denotes that these are specific to the *sparse* version.

- `SM_CONTENT_S`

  This routine gives access to the contents of the sparse `SUNMatrix`.

  The assignment `A_cont = SM_CONTENT_S(A)` sets `A_cont` to be a pointer to the sparse `SUNMatrix` content structure.

  Implementation:

  ```
  #define SM_CONTENT_S(A)     ( (SUNMatrixContent_Sparse)(A->content) )
  ```

- `SM_ROWS_S`, `SM_COLUMNS_S`, `SM_NNZ_S`, `SM_NP_S`, and `SM_SPARSETYPE_S`

  These macros give individual access to various lengths relevant to the content of a sparse `SUNMatrix`.

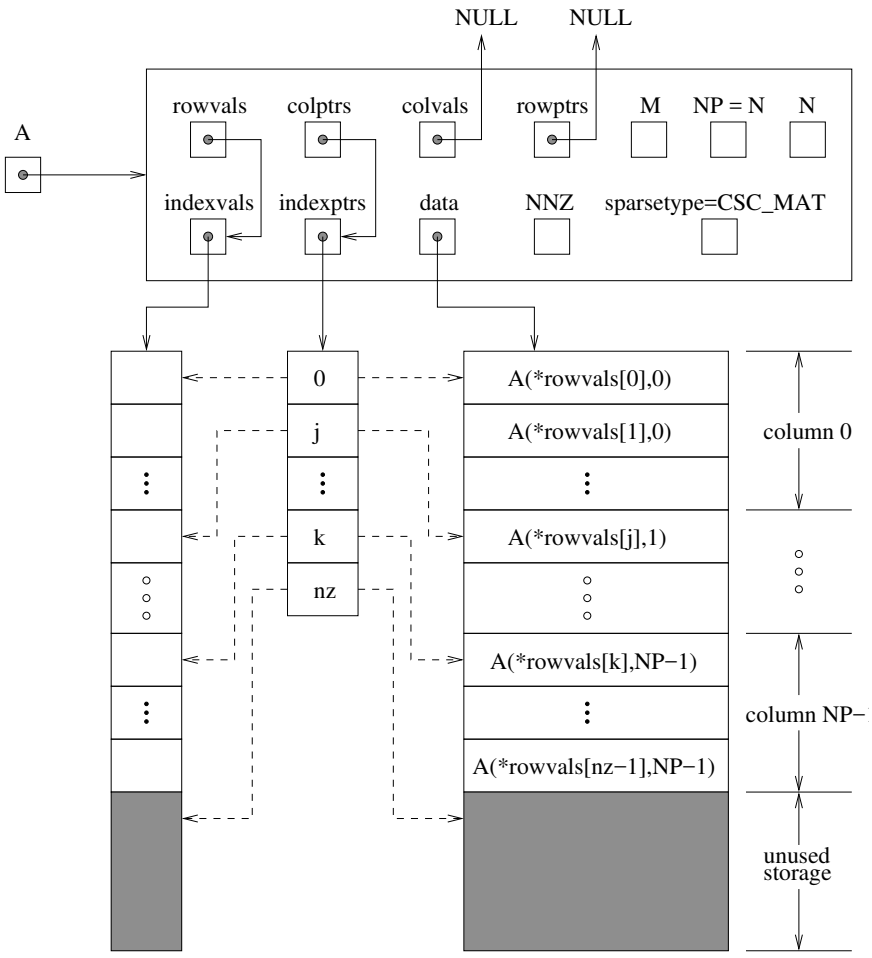


Figure 7.2: Diagram of the storage for a compressed-sparse-column matrix. Here `A` is an $\texttt{M} \times \texttt{N}$ sparse matrix with storage for up to `NNZ` nonzero entries (the allocated length of both `data` and `indexvals`). The entries in `indexvals` may assume values from 0 to $\texttt{M} - 1$, corresponding to the row index (zero-based) of each nonzero value. The entries in `data` contain the values of the nonzero entries, with the row i, column j entry of `A` (again, zero-based) denoted as `A(i,j)`. The `indexptrs` array contains $\texttt{N}+1$ entries; the first `N` denote the starting index of each column within the `indexvals` and `data` arrays, while the final entry points one past the final nonzero entry. Here, although `NNZ` values are allocated, only `nz` are actually filled in; the greyed-out portions of `data` and `indexvals` indicate extra allocated space.

These may be used either to retrieve or to set these values. For example, the assignment `A_rows = SM_ROWS_S(A)` sets `A_rows` to be the number of rows in the matrix `A`. Similarly, the assignment `SM_COLUMNS_S(A) = A_cols` sets the number of columns in `A` to equal `A_cols`.

Implementation:

```
#define SM_ROWS_S(A)        ( SM_CONTENT_S(A)->M )
#define SM_COLUMNS_S(A)     ( SM_CONTENT_S(A)->N )
#define SM_NNZ_S(A)         ( SM_CONTENT_S(A)->NNZ )
#define SM_NP_S(A)          ( SM_CONTENT_S(A)->NP )
#define SM_SPARSETYPE_S(A)  ( SM_CONTENT_S(A)->sparsetype )
```

- `SM_DATA_S`, `SM_INDEXVALS_S`, and `SM_INDEXPTRS_S`

  These macros give access to the `data` and index arrays for the matrix entries.

  The assignment `A_data = SM_DATA_S(A)` sets `A_data` to be a pointer to the first component of the data array for the sparse `SUNMatrix A`. The assignment `SM_DATA_S(A) = A_data` sets the data array of `A` to be `A_data` by storing the pointer `A_data`.

  Similarly, the assignment `A_indexvals = SM_INDEXVALS_S(A)` sets `A_indexvals` to be a pointer to the array of index values (i.e. row indices for a CSC matrix, or column indices for a CSR matrix) for the sparse `SUNMatrix A`. The assignment `A_indexptrs = SM_INDEXPTRS_S(A)` sets `A_indexptrs` to be a pointer to the array of index pointers (i.e. the starting indices in the data/indexvals arrays for each row or column in CSR or CSC formats, respectively).

  Implementation:

```
#define SM_DATA_S(A)        ( SM_CONTENT_S(A)->data )
#define SM_INDEXVALS_S(A)   ( SM_CONTENT_S(A)->indexvals )
#define SM_INDEXPTRS_S(A)   ( SM_CONTENT_S(A)->indexptrs )
```

The SUNMATRIX_SPARSE module defines sparse implementations of all matrix operations listed in Table 7.2. Their names are obtained from those in Table 7.2 by appending the suffix `_Sparse` (e.g. `SUNMatCopy_Sparse`). The module SUNMATRIX_SPARSE provides the following additional user-callable routines:

- `SUNSparseMatrix`

  This function creates and allocates memory for a sparse `SUNMatrix`. Its arguments are the number of rows and columns of the matrix, `M` and `N`, the maximum number of nonzeros to be stored in the matrix, `NNZ`, and a flag `sparsetype` indicating whether to use CSR or CSC format (valid arguments are `CSR_MAT` or `CSC_MAT`).

```
SUNMatrix SUNSparseMatrix(sunindextype M, sunindextype N,
                          sunindextype NNZ, int sparsetype);
```

- `SUNSparseFromDenseMatrix`

  This function creates a new sparse matrix from an existing dense matrix by copying all values with magnitude larger than `droptol` into the sparse matrix structure.

  Requirements:

  – `A` must have type `SUNMATRIX_DENSE`;

  – `droptol` must be non-negative;

  – `sparsetype` must be either `CSC_MAT` or `CSR_MAT`.

The function returns NULL if any requirements are violated, or if the matrix storage request cannot be satisfied.

```
SUNMatrix SUNSparseFromDenseMatrix(SUNMatrix A, realtype droptol,
                                   int sparsetype);
```

- `SUNSparseFromBandMatrix`

  This function creates a new sparse matrix from an existing band matrix by copying all values with magnitude larger than `droptol` into the sparse matrix structure.

  Requirements:

  - `A` must have type `SUNMATRIX_BAND`;
  - `droptol` must be non-negative;
  - `sparsetype` must be either `CSC_MAT` or `CSR_MAT`.

  The function returns NULL if any requirements are violated, or if the matrix storage request cannot be satisfied.

  ```
  SUNMatrix SUNSparseFromBandMatrix(SUNMatrix A, realtype droptol,
                                    int sparsetype);
  ```

- `SUNSparseMatrix_Realloc`

  This function reallocates internal storage arrays in a sparse matrix so that the resulting sparse matrix has no wasted space (i.e. the space allocated for nonzero entries equals the actual number of nonzeros, `indexptrs[NP]`). Returns 0 on success and 1 on failure (e.g. if the input matrix is not sparse).

  ```
  int SUNSparseMatrix_Realloc(SUNMatrix A);
  ```

- `SUNSparseMatrix_Print`

  This function prints the content of a sparse `SUNMatrix` to the output stream specified by `outfile`. Note: `stdout` or `stderr` may be used as arguments for `outfile` to print directly to standard output or standard error, respectively.

  ```
  void SUNSparseMatrix_Print(SUNMatrix A, FILE* outfile);
  ```

- `SUNSparseMatrix_Rows`

  This function returns the number of rows in the sparse `SUNMatrix`.

  ```
  sunindextype SUNSparseMatrix_Rows(SUNMatrix A);
  ```

- `SUNSparseMatrix_Columns`

  This function returns the number of columns in the sparse `SUNMatrix`.

  ```
  sunindextype SUNSparseMatrix_Columns(SUNMatrix A);
  ```

- `SUNSparseMatrix_NNZ`

  This function returns the number of entries allocated for nonzero storage for the sparse matrix `SUNMatrix`.

  ```
  sunindextype SUNSparseMatrix_NNZ(SUNMatrix A);
  ```

- `SUNSparseMatrix_NP`

  This function returns the number of columns/rows for the sparse `SUNMatrix`, depending on whether the matrix uses CSC/CSR format, respectively. The `indexptrs` array has `NP+1` entries.

  ```
  sunindextype SUNSparseMatrix_NP(SUNMatrix A);
  ```

- SUNSparseMatrix_SparseType

  This function returns the storage type (CSR_MAT or CSC_MAT) for the sparse SUNMatrix.

  ```
  int SUNSparseMatrix_SparseType(SUNMatrix A);
  ```

- SUNSparseMatrix_Data

  This function returns a pointer to the data array for the sparse SUNMatrix.

  ```
  realtype* SUNSparseMatrix_Data(SUNMatrix A);
  ```

- SUNSparseMatrix_IndexValues

  This function returns a pointer to index value array for the sparse SUNMatrix: for CSR format this is the column index for each nonzero entry, for CSC format this is the row index for each nonzero entry.

  ```
  sunindextype* SUNSparseMatrix_IndexValues(SUNMatrix A);
  ```

- SUNSparseMatrix_IndexPointers

  This function returns a pointer to the index pointer array for the sparse SUNMatrix: for CSR format this is the location of the first entry of each row in the data and indexvalues arrays, for CSC format this is the location of the first entry of each column.

  ```
  sunindextype* SUNSparseMatrix_IndexPointers(SUNMatrix A);
  ```


Within the SUNMatMatvec_Sparse routine, internal consistency checks are performed to ensure that the matrix is called with consistent NVECTOR implementations. These are currently limited to: NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS. As additional compatible vector implementations are added to SUNDIALS, these will be included within this compatibility check.

For solvers that include a Fortran interface module, the SUNMATRIX_SPARSE module also includes the Fortran-callable function `FSUNSparseMatInit(code, M, N, NNZ, sparsetype, ier)` to initialize this SUNMATRIX_SPARSE module for a given SUNDIALS solver. Here `code` is an integer input for the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); `M`, `N` and `NNZ` are the corresponding sparse matrix construction arguments (declared to match C type `long int`); `sparsetype` is an integer flag indicating the sparse storage type (0 for CSC, 1 for CSR); and `ier` is an error return flag equal to 0 for success and -1 for failure. Each of `code`, `sparsetype` and `ier` are declared so as to match C type `int`. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function `FSUNSparseMassMatInit(M, N, NNZ, sparsetype, ier)` initializes this SUNMATRIX_SPARSE module for storing the mass matrix.

7.4 SUNMatrix Examples

There are SUNMatrix examples that may be installed for each implementation: dense, banded, and sparse. Each implementation makes use of the functions in `test_sunmatrix.c`. These example functions show simple usage of the SUNMatrix family of functions. The inputs to the examples depend on the matrix type, and are output to `stdout` if the example is run without the appropriate number of command-line arguments.
The following is a list of the example functions in `test_sunmatrix.c`:

- `Test_SUNMatGetID`: Verifies the returned matrix ID against the value that should be returned.
- `Test_SUNMatClone`: Creates clone of an existing matrix, copies the data, and checks that their values match.
- `Test_SUNMatZero`: Zeros out an existing matrix and checks that each entry equals 0.0.
- `Test_SUNMatCopy`: Clones an input matrix, copies its data to a clone, and verifies that all values match.

- `Test_SUNMatScaleAdd`: Given an input matrix A and an input identity matrix I, this test clones and copies A to a new matrix B, computes $B = -B + B$, and verifies that the resulting matrix entries equal 0.0. Additionally, if the matrix is square, this test clones and copies A to a new matrix D, clones and copies I to a new matrix C, computes $D = D + I$ and $C = C + A$ using `SUNMatScaleAdd`, and then verifies that $C == D$.

- `Test_SUNMatScaleAddI`: Given an input matrix A and an input identity matrix I, this clones and copies I to a new matrix B, computes $B = -B + I$ using `SUNMatScaleAddI`, and verifies that the resulting matrix entries equal 0.0.

- `Test_SUNMatMatvec` Given an input matrix A and input vectors x and y such that $y = Ax$, this test has different behavior depending on whether A is square. If it is square, it clones and copies A to a new matrix B, computes $B = 3B + I$ using `SUNMatScaleAddI`, clones y to new vectors w and z, computes $z = Bx$ using `SUNMatMatvec`, computes $w = 3y + x$ using `N_VLinearSum`, and verifies that $w == z$. If A is not square, it just clones y to a new vector z, computes $z = Ax$ using `SUNMatMatvec`, and verifies that $y == z$.

- `Test_SUNMatSpace` verifies that `SUNMatSpace` can be called, and outputs the results to `stdout`.

7.5 SUNMatrix functions used by KINSOL

In Table 7.4 below, we list the matrix functions in the SUNMATRIX module used within the KINSOL package. The table also shows, for each function, which of the code modules uses the function. Neither the main KINSOL integrator or the KINSPILS interface call SUNMATRIX functions directly, so the table columns are specific to the KINDLS direct solver interface and the KINBBDPRE preconditioner module.

At this point, we should emphasize that the KINSOL user does not need to know anything about the usage of matrix functions by the KINSOL code modules in order to use KINSOL. The information is presented as an implementation detail for the interested reader.

Table 7.4: List of matrix functions usage by KINSOL code modules

	KINDLS	KINBBDPRE
SUNMatGetID	✓	
SUNMatDestroy		✓
SUNMatZero	✓	✓
SUNMatSpace		†

The matrix functions listed in Table 7.2 with a † symbol are optionally used, in that these are only called if they are implemented in the SUNMATRIX module that is being used (i.e. their function pointers are non-`NULL`). The matrix functions listed in Table 7.2 that are *not* used by KINSOL are: `SUNMatCopy`, `SUNMatClone`, `SUNMatScaleAdd`, `SUNMatScaleAddI` and `SUNMatMatvec`. Therefore a user-supplied SUNMATRIX module for KINSOL could omit these functions.

Chapter 8

Description of the SUNLinearSolver module

For problems that involve the solution of linear systems of equations, the SUNDIALS solvers operate using generic linear solver modules (of type `SUNLinearSolver`), through a set of operations defined by the particular SUNLINSOL implementation. These work in coordination with the SUNDIALS generic NVECTOR and SUNMATRIX modules to provide a set of compatible data structures and solvers for the solution of linear systems using direct or iterative methods. Moreover, users can provide their own specific SUNLINSOL implementation to each SUNDIALS solver, particularly in cases where they provide their own NVECTOR and/or SUNMATRIX modules, and the customized linear solver leverages these additional data structures to create highly efficient and/or scalable solvers for their particular problem. Additionally, SUNDIALS provides native implementations SUNLINSOL modules, as well as SUNLINSOL modules that interface between SUNDIALS and external linear solver libraries.

The various SUNDIALS solvers have been designed to specifically leverage the use of either *direct linear solvers* or *scaled, preconditioned, iterative linear solvers*, through their "Dls" and "Spils" interfaces, respectively. Additionally, SUNDIALS solvers can make use of user-supplied custom linear solvers, whether these are problem-specific or come from external solver libraries.

For iterative (and possibly custom) linear solvers, the SUNDIALS solvers leverage scaling and preconditioning, as applicable, to balance error between solution components and to accelerate convergence of the linear solver. To this end, instead of solving the linear system $Ax = b$ directly, we apply the underlying iterative algorithm to the transformed system

$$\tilde{A}\tilde{x} = \tilde{b} \tag{8.1}$$

where

$$\begin{aligned} \tilde{A} &= S_1 P_1^{-1} A P_2^{-1} S_2^{-1}, \\ \tilde{b} &= S_1 P_1^{-1} b, \\ \tilde{x} &= S_2 P_2 x, \end{aligned} \tag{8.2}$$

and where

- P_1 is the left preconditioner,
- P_2 is the right preconditioner,
- S_1 is a diagonal matrix of scale factors for $P_1^{-1}b$,
- S_2 is a diagonal matrix of scale factors for $P_2 x$.

The SUNDIALS solvers request that iterative linear solvers stop based on the 2-norm of the scaled preconditioned residual meeting a prescribed tolerance

$$\left\| \tilde{b} - \tilde{A}\tilde{x} \right\|_2 < \text{tol}.$$

We note that not all of the iterative linear solvers implemented in SUNDIALS support the full range of the above options. Similarly, some of the SUNDIALS integrators only utilize a subset of these options. Exceptions to the operators shown above are described in the documentation for each SUNLINSOL implementation, or for each SUNDIALS solver "Spils" interface.

The generic `SUNLinearSolver` type has been modeled after the object-oriented style of the generic `N_Vector` type. Specifically, a generic `SUNLinearSolver` is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the linear solver, and an *ops* field pointing to a structure with generic linear solver operations. The type `SUNLinearSolver` is defined as

```
typedef struct _generic_SUNLinearSolver *SUNLinearSolver;

struct _generic_SUNLinearSolver {
  void *content;
  struct _generic_SUNLinearSolver_Ops *ops;
};
```

The `_generic_SUNLinearSolver_Ops` structure is essentially a list of pointers to the various actual linear solver operations, and is defined as

```
struct _generic_SUNLinearSolver_Ops {
  SUNLinearSolver_Type (*gettype)(SUNLinearSolver);
  int                  (*setatimes)(SUNLinearSolver, void*, ATimesFn);
  int                  (*setpreconditioner)(SUNLinearSolver, void*,
                                            PSetupFn, PSolveFn);
  int                  (*setscalingvectors)(SUNLinearSolver,
                                            N_Vector, N_Vector);
  int                  (*initialize)(SUNLinearSolver);
  int                  (*setup)(SUNLinearSolver, SUNMatrix);
  int                  (*solve)(SUNLinearSolver, SUNMatrix, N_Vector,
                                N_Vector, realtype);
  int                  (*numiters)(SUNLinearSolver);
  realtype             (*resnorm)(SUNLinearSolver);
  long int             (*lastflag)(SUNLinearSolver);
  int                  (*space)(SUNLinearSolver, long int*, long int*);
  N_Vector             (*resid)(SUNLinearSolver);
  int                  (*free)(SUNLinearSolver);
};
```

The generic SUNLINSOL module defines and implements the linear solver operations acting on `SUNLinearSolver` objects. These routines are in fact only wrappers for the linear solver operations defined by a particular SUNLINSOL implementation, which are accessed through the *ops* field of the `SUNLinearSolver` structure. To illustrate this point we show below the implementation of a typical linear solver operation from the generic SUNLINSOL module, namely `SUNLinSolInitialize`, which initializes a SUNLINSOL object for use after it has been created and configured, and returns a flag denoting a successful/failed operation:

```
int SUNLinSolInitialize(SUNLinearSolver S)
{
  return ((int) S->ops->initialize(S));
}
```

Table 8.2 contains a complete list of all linear solver operations defined by the generic SUNLINSOL module. In order to support both direct and iterative linear solver types, the generic SUNLINSOL module defines linear solver routines (or arguments) that may be specific to individual use cases. As such, for each routine we specify its intended use. If a custom SUNLINSOL module is provided, the function pointers for non-required routines may be set to `NULL` to indicate that they are not provided.

A particular implementation of the SUNLINSOL module must:

Table 8.1: Identifiers associated with linear solver kernels supplied with SUNDIALS.

Linear Solver ID	Solver type	ID Value
SUNLINEARSOLVER_DIRECT	Direct solvers	0
SUNLINEARSOLVER_ITERATIVE	Iterative solvers	1
SUNLINEARSOLVER_CUSTOM	Custom solvers	2

- Specify the *content* field of the `SUNLinearSolver` object.
- Define and implement a minimal subset of the linear solver operations. See the documentation for each SUNDIALS linear solver interface to determine which SUNLINSOL operations they require.

  Note that the names of these routines should be unique to that implementation in order to permit using more than one SUNLINSOL module (each with different `SUNLinearSolver` internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free a `SUNLinearSolver` with the new *content* field and with *ops* pointing to the new linear solver operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined `SUNLinearSolver` (e.g., routines to set various configuration options for tuning the linear solver to a particular problem).
- Optionally, provide functions as needed for that particular implementation to access different parts in the *content* field of the newly defined `SUNLinearSolver` object (e.g., routines to return various statistics from the solver).

Each SUNLINSOL implementation included in SUNDIALS has a "type" identifier specified in enumeration and shown in Table 8.1. It is recommended that a user-supplied SUNLINSOL implementation set this identifier based on the SUNDIALS solver interface they intend to use: "Dls" interfaces require the `SUNLINEARSOLVER_DIRECT` SUNLINSOL objects and "Spils" interfaces require the `SUNLINEARSOLVER_ITERATIVE` objects.

Table 8.2: Description of the `SUNLinearSolver` operations

Name	Usage and Description
`SUNLinSolGetType`	`type = SUNLinSolGetType(LS);` Returns the type identifier for the linear solver `LS`. It is used to determine the solver type (direct, iterative, or custom) from the abstract `SUNLinearSolver` interface. This is used to assess compatibility with SUNDIALS-provided linear solver interfaces. Returned values are given in the Table 8.1.

continued on next page

Name	Usage and Description
SUNLinSolInitialize	`ier = SUNLinSolInitialize(LS);` Performs linear solver initialization (assumes that all solver-specific options have been set). This should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 8.4.
SUNLinSolSetup	`ier = SUNLinSolSetup(LS, A);` Performs any linear solver setup needed, based on an updated system SUNMATRIX `A`. This may be called frequently (e.g. with a full Newton method) or infrequently (for a modified Newton method), based on the type of integrator and/or nonlinear solver requesting the solves. This should return zero for a successful call, a positive value for a recoverable failure and a negative value for an unrecoverable failure, ideally returning one of the generic error codes listed in Table 8.4.
SUNLinSolSolve	`ier = SUNLinSolSolve(LS, A, x, b, tol);` Solves a linear system $Ax = b$. This should return zero for a successful call, a positive value for a recoverable failure and a negative value for an unrecoverable failure, ideally returning one of the generic error codes listed in Table 8.4. **Direct solvers:** can ignore the `realtype` argument `tol`. **Iterative solvers:** can ignore the SUNMATRIX input `A` since a `NULL` argument will be passed (these should instead rely on the matrix-vector product function supplied through the routine `SUNLinSolSetATimes`). These should attempt to solve to the specified `realtype` tolerance `tol` in a weighted 2-norm. If the solver does not support scaling then it should just use a 2-norm. **Custom solvers:** all arguments will be supplied, and if the solver is approximate then it should attempt to solve to the specified `realtype` tolerance `tol` in a weighted 2-norm. If the solver does not support scaling then it should just use a 2-norm.
SUNLinSolFree	`ier = SUNLinSolFree(LS);` Frees memory allocated by the linear solver. This should return zero for a successful call, and a negative value for a failure.
SUNLinSolSetATimes	`ier = SUNLinSolSetATimes(LS, A_data, ATimes);` (Iterative/Custom linear solvers only) Provides `ATimesFn` function pointer, as well as a `void *` pointer to a data structure used by this routine, to a linear solver object. SUNDIALS solvers will call this function to set the matrix-vector product function to either a solver-provided difference-quotient via vector operations or a user-supplied solver-specific routine. This routine should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 8.4.

continued on next page

Name	Usage and Description
SUNLinSolSetPreconditioner	`ier = SUNLinSolSetPreconditioner(LS, Pdata, Pset, Psol);` (Optional; Iterative/Custom linear solvers only) Provides `PSetupFn` and `PSolveFn` function pointers that implement the preconditioner solves P_1^{-1} and P_2^{-1} from equations (8.1)-(8.2). This routine will be called by a SUNDIALS solver, which will provide translation between the generic `Pset` and `Psol` calls and the integrator-specific and integrator- or user-supplied routines. This routine should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 8.4.
SUNLinSolSetScalingVectors	`ier = SUNLinSolSetScalingVectors(LS, s1, s2);` (Optional; Iterative/Custom linear solvers only) Sets pointers to left/right scaling vectors for the linear system solve. Here, `s1` is an NVECTOR of positive scale factors containing the diagonal of the matrix S_1 from equations (8.1)-(8.2). Similarly, `s2` is an NVECTOR containing the diagonal of S_2 from equations (8.1)-(8.2). Neither of these vectors are tested for positivity, and a `NULL` argument for either indicates that the corresponding scaling matrix is the identity. This routine should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 8.4.
SUNLinSolNumIters	`its = SUNLinSolNumIters(LS);` (Optional; Iterative/Custom linear solvers only) Should return the `int` number of linear iterations performed in the last 'solve' call.
SUNLinSolResNorm	`rnorm = SUNLinSolResNorm(LS);` (Optional; Iterative/Custom linear solvers only) Should return the `realtype` final residual norm from the last 'solve' call.
SUNLinSolResid	`rvec = SUNLinSolResid(LS);` (Optional; Iterative/Custom linear solvers only) If an iterative method computes the preconditioned initial residual and returns with a successful solve without performing any iterations (i.e. either the initial guess or the preconditioner is sufficiently accurate), then this function may be called by the SUNDIALS solver. This routine should return the NVECTOR containing the preconditioned initial residual vector.

continued on next page

Name	Usage and Description
SUNLinLastFlag	lflag = SUNLinLastFlag(LS); (Optional) Should return the last error flag encountered within the linear solver. This is not called by the SUNDIALS solvers directly; it allows the user to investigate linear solver issues after a failed solve.
SUNLinSolSpace	ier = SUNLinSolSpace(LS, &lrw, &liw); (Optional) Returns the storage requirements for the linear solver LS. lrw is a long int containing the number of realtype words and liw is a long int containing the number of integer words. The return value is an integer flag denoting success/failure of the operation. This function is advisory only, for use in determining a user's total space requirements.

8.1 Description of the client-supplied SUNLinearSolver routines

The SUNDIALS packages provide the ATimes, Pset and Psol routines utilized by the SUNLINSOL modules. These function types are defined in the header file sundials/sundials_iterative.h, and are described here in case a user wishes to interact directly with an iterative SUNLINSOL object.

ATimesFn

Definition `typedef int (*ATimesFn)(void *A_data, N_Vector v, N_Vector z);`

Purpose These functions compute the action of a matrix on a vector, performing the operation $z = Av$. Memory for z should already be allocted prior to calling this function. The vector v should be left unchanged.

Arguments A_data is a pointer to client data, the same as that supplied to SUNLinSolSetATimes.

v is the input vector to multiply.

z is the output vector computed.

Return value This routine should return 0 if successful and a non-zero value if unsuccessful.

Notes

PSetupFn

Definition `typedef int (*PSetupFn)(void *P_data)`

Purpose These functions set up any requisite problem data in preparation for calls to the corresponding PSolveFn.

Arguments P_data is a pointer to client data, the same pointer as that supplied to the routine SUNLinSolSetPreconditioner.

Return value This routine should return 0 if successful and a non-zero value if unsuccessful.

Notes

`PSolveFn`

Definition
```
typedef int (*PSolveFn)(void *P_data, N_Vector r, N_Vector z,
                        realtype tol, int lr)
```

Purpose These functions solve the preconditioner equation $Pz = r$ for the vector z. Memory for `z` should already be allocted prior to calling this function. The parameter `P_data` is a pointer to any information about P which the function needs in order to do its job (set up by the corresponding `PSetupFn`. The parameter `lr` is input, and indicates whether P is to be taken as the left preconditioner or the right preconditioner: `lr` $= 1$ for left and `lr` $= 2$ for right. If preconditioning is on one side only, `lr` can be ignored. If the preconditioner is iterative, then it should strive to solve the preconditioner equation so that

$$\|Pz - r\|_{\text{wrms}} < tol$$

where the weight vector for the WRMS norm may be accessed from the main package memory structure. The vector `r` should not be modified by the `PSolveFn`.

Arguments `P_data` is a pointer to client data, the same pointer as that supplied to the routine `SUNLinSolSetPreconditioner`.

`r` is the right-hand side vector for the preconditioner system

`z` is the solution vector for the preconditioner system

`tol` is the desired tolerance for an iterative preconditioner

`lr` is flag indicating whether the routine should perform left (1) or right (2) preconditioning.

Return value This routine should return 0 if successful and a non-zero value if unsuccessful. On a failure, a negative return value indicates an unrecoverable condition, while a positive value indicates a recoverable one, in which the calling routine may reattempt the solution after updating preconditioner data.

Notes

8.2 Compatibility of SUNLinearSolver modules

We note that not all SUNLINSOL types are compatible with all SUNMATRIX and NVECTOR types provided with SUNDIALS. In Table 8.3 we show the direct linear solvers available as SUNLINSOL modules, and the compatible matrix implementations. Recall that Table 4.1 shows the compatibility between all SUNLINSOL modules and vector implementations.

Table 8.3: SUNDIALS direct linear solvers and matrix implementations that can be used for each.

Linear Solver Interface	Dense Matrix	Banded Matrix	Sparse Matrix	User Supplied
Dense	✓			✓
Band		✓		✓
LapackDense	✓			✓
LapackBand		✓		✓
KLU			✓	✓
SUPERLUMT			✓	✓

continued on next page

Linear Solver Interface	Dense Matrix	Banded Matrix	Sparse Matrix	User Supplied
User supplied	✓	✓	✓	✓

The functions within the SUNDIALS-provided `SUNLinearSolver` implementations return a common set of error codes, shown below in the Table 8.4.

Table 8.4: Description of the `SUNLinearSolver` error codes

Name	Value	Description
SUNLS_SUCCESS	0	successful call or converged solve
SUNLS_MEM_NULL	-1	the memory argument to the function is `NULL`
SUNLS_ILL_INPUT	-2	an illegal input has been provided to the function
SUNLS_MEM_FAIL	-3	failed memory access or allocation
SUNLS_ATIMES_FAIL_UNREC	-4	an unrecoverable failure occurred in the `ATimes` routine
SUNLS_PSET_FAIL_UNREC	-5	an unrecoverable failure occurred in the `Pset` routine
SUNLS_PSOLVE_FAIL_UNREC	-6	an unrecoverable failure occurred in the `Psolve` routine
SUNLS_PACKAGE_FAIL_UNREC	-7	an unrecoverable failure occurred in an external linear solver package
SUNLS_GS_FAIL	-8	a failure occurred during Gram-Schmidt orthogonalization (SUNLINSOL_SPGMR/SUNLINSOL_SPFGMR)
SUNLS_QRSOL_FAIL	-9	a singular R matrix was encountered in a QR factorization (SUNLINSOL_SPGMR/SUNLINSOL_SPFGMR)
SUNLS_RES_REDUCED	1	an iterative solver reduced the residual, but did not converge to the desired tolerance
SUNLS_CONV_FAIL	2	an iterative solver did not converge (and the residual was not reduced)
SUNLS_ATIMES_FAIL_REC	3	a recoverable failure occurred in the `ATimes` routine
SUNLS_PSET_FAIL_REC	4	a recoverable failure occurred in the `Pset` routine
SUNLS_PSOLVE_FAIL_REC	5	a recoverable failure occurred in the `Psolve` routine
SUNLS_PACKAGE_FAIL_REC	6	a recoverable failure occurred in an external linear solver package
SUNLS_QRFACT_FAIL	7	a singular matrix was encountered during a QR factorization (SUNLINSOL_SPGMR/SUNLINSOL_SPFGMR)
SUNLS_LUFACT_FAIL	8	a singular matrix was encountered during a LU factorization (SUNLINSOL_DENSE/SUNLINSOL_BAND)

8.3 The SUNLinearSolver_Dense implementation

The dense implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_DENSE, is designed to be used with the corresponding SUNMATRIX_DENSE matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR_SERIAL, NVECTOR_OPENMP or NVECTOR_PTHREADS). The SUNLINSOL_DENSE module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_Dense {
  sunindextype N;
  sunindextype *pivots;
```

```
  long int last_flag;
};
```

These entries of the *content* field contain the following information:

N - size of the linear system,

pivots - index array for partial pivoting in LU factorization,

last_flag - last error return flag from internal function evaluations.

This solver is constructed to perform the following operations:

- The "setup" call performs a LU factorization with partial (row) pivoting ($\mathcal{O}(N^3)$ cost), $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix with 1's on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX_DENSE object A, with pivoting information encoding P stored in the `pivots` array.
- The "solve" call performs pivoting and forward and backward substitution using the stored `pivots` array and the LU factors held in the SUNMATRIX_DENSE object ($\mathcal{O}(N^2)$ cost).

The header file to include when using this module is `sunlinsol/sunlinsol_dense.h`. The SUNLINSOL_DENSE module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsoldense` module library.

The SUNLINSOL_DENSE module defines dense implementations of all "direct" linear solver operations listed in Table 8.2:

- `SUNLinSolGetType_Dense`
- `SUNLinSolInitialize_Dense` – this does nothing, since all consistency checks are performed at solver creation.
- `SUNLinSolSetup_Dense` – this performs the LU factorization.
- `SUNLinSolSolve_Dense` – this uses the LU factors and `pivots` array to perform the solve.
- `SUNLinSolLastFlag_Dense`
- `SUNLinSolSpace_Dense` – this only returns information for the storage *within* the solver object, i.e. storage for `N`, `last_flag`, and `pivots`.
- `SUNLinSolFree_Dense`

The module SUNLINSOL_DENSE provides the following additional user-callable constructor routine:

- `SUNDenseLinearSolver`

  This function creates and allocates memory for a dense `SUNLinearSolver`. Its arguments are an NVECTOR and SUNMATRIX, that it uses to determine the linear system size and to assess compatibility with the linear solver implementation.

  This routine will perform consistency checks to ensure that it is called with consistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX_DENSE matrix type and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

  If either `A` or `y` are incompatible then this routine will return `NULL`.

  ```
  SUNLinearSolver SUNDenseLinearSolver(N_Vector y, SUNMatrix A);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_DENSE module also includes the Fortran-callable function `FSUNDenseLinSolInit(code, ier)` to initialize this SUNLINSOL_DENSE module for a given SUNDIALS solver. Here `code` is an integer input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); `ier` is an error return flag equal to 0 for success and -1 for failure. Both `code` and `ier` are declared to match C type `int`. This routine must be called *after* both the NVECTOR and SUNMATRIX objects have been initialized. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function `FSUNMassDenseLinSolInit(ier)` initializes this SUNLINSOL_DENSE module for solving mass matrix linear systems.

8.4 The SUNLinearSolver_Band implementation

The band implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_BAND, is designed to be used with the corresponding SUNMATRIX_BAND matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR_SERIAL, NVECTOR_OPENMP or NVECTOR_PTHREADS). The SUNLINSOL_BAND module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_Band {
  sunindextype N;
  sunindextype *pivots;
  long int last_flag;
};
```

These entries of the *content* field contain the following information:

N - size of the linear system,

pivots - index array for partial pivoting in LU factorization,

last_flag - last error return flag from internal function evaluations.

This solver is constructed to perform the following operations:

- The "setup" call performs a LU factorization with partial (row) pivoting, $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix with 1's on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX_BAND object A, with pivoting information encoding P stored in the `pivots` array.
- The "solve" call performs pivoting and forward and backward substitution using the stored `pivots` array and the LU factors held in the SUNMATRIX_BAND object.

- A must be allocated to accommodate the increase in upper bandwidth that occurs during factorization. More precisely, if A is a band matrix with upper bandwidth `mu` and lower bandwidth `ml`, then the upper triangular factor U can have upper bandwidth as big as `smu = MIN(N-1,mu+ml)`. The lower triangular factor L has lower bandwidth `ml`.

The header file to include when using this module is `sunlinsol/sunlinsol_band.h`. The SUNLINSOL_BAND module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolband` module library.
The SUNLINSOL_BAND module defines band implementations of all "direct" linear solver operations listed in Table 8.2:

- `SUNLinSolGetType_Band`
- `SUNLinSolInitialize_Band` – this does nothing, since all consistency checks are performed at solver creation.
- `SUNLinSolSetup_Band` – this performs the LU factorization.

- SUNLinSolSolve_Band – this uses the *LU* factors and pivots array to perform the solve.
- SUNLinSolLastFlag_Band
- SUNLinSolSpace_Band – this only returns information for the storage *within* the solver object, i.e. storage for N, last_flag, and pivots.
- SUNLinSolFree_Band

The module SUNLINSOL_BAND provides the following additional user-callable constructor routine:

- SUNBandLinearSolver

  This function creates and allocates memory for a band SUNLinearSolver. Its arguments are an NVECTOR and SUNMATRIX, that it uses to determine the linear system size and to assess compatibility with the linear solver implementation.

  This routine will perform consistency checks to ensure that it is called with consistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX_BAND matrix type and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

  Additionally, this routine will verify that the input matrix A is allocated with appropriate upper bandwidth storage for the *LU* factorization.

  If either A or y are incompatible then this routine will return NULL.

  ```
  SUNLinearSolver SUNBandLinearSolver(N_Vector y, SUNMatrix A);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_BAND module also includes the Fortran-callable function FSUNBandLinSolInit(code, ier) to initialize this SUNLINSOL_BAND module for a given SUNDIALS solver. Here code is an integer input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); ier is an error return flag equal to 0 for success and -1 for failure. Both code and ier are declared to match C type int. This routine must be called *after* both the NVECTOR and SUNMATRIX objects have been initialized. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function FSUNMassBandLinSolInit(ier) initializes this SUNLINSOL_BAND module for solving mass matrix linear systems.

8.5 The SUNLinearSolver_LapackDense implementation

The LAPACK dense implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_LAPACKDENSE, is designed to be used with the corresponding SUNMATRIX_DENSE matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, or NVECTOR_PTHREADS). The SUNLINSOL_LAPACKDENSE module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_Dense {
  sunindextype N;
  sunindextype *pivots;
  long int last_flag;
};
```

These entries of the *content* field contain the following information:

N - size of the linear system,

pivots - index array for partial pivoting in LU factorization,

last_flag - last error return flag from internal function evaluations.

The SUNLINSOL_LAPACKDENSE module is a SUNLINSOL wrapper for the LAPACK dense matrix factorization and solve routines, `*GETRF` and `*GETRS`, where `*` is either `D` or `S`, depending on whether SUNDIALS was configured to have `realtype` set to `double` or `single`, respectively (see Section 4.2). In order to use the SUNLINSOL_LAPACKDENSE module it is assumed that LAPACK has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with LAPACK (see Appendix A for details). We note that since there do not exist 128-bit floating-point factorization and solve routines in LAPACK, this interface cannot be compiled when using `extended` precision for `realtype`. Similarly, since there do not exist 64-bit integer LAPACK routines, the SUNLINSOL_LAPACKDENSE module also cannot be compiled when using `int64_t` for the `sunindextype`.

This solver is constructed to perform the following operations:

- The "setup" call performs a LU factorization with partial (row) pivoting ($\mathcal{O}(N^3)$ cost), $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix with 1's on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX_DENSE object A, with pivoting information encoding P stored in the `pivots` array.
- The "solve" call performs pivoting and forward and backward substitution using the stored `pivots` array and the LU factors held in the SUNMATRIX_DENSE object ($\mathcal{O}(N^2)$ cost).

The header file to include when using this module is `sunlinsol/sunlinsol_lapackdense.h`. The installed module library to link to is `libsundials_sunlinsollapackdense.`*`lib`* where *`.lib`* is typically `.so` for shared libraries and `.a` for static libraries.

The SUNLINSOL_LAPACKDENSE module defines dense implementations of all "direct" linear solver operations listed in Table 8.2:

- `SUNLinSolGetType_LapackDense`
- `SUNLinSolInitialize_LapackDense` – this does nothing, since all consistency checks are performed at solver creation.
- `SUNLinSolSetup_LapackDense` – this calls either `DGETRF` or `SGETRF` to perform the LU factorization.
- `SUNLinSolSolve_LapackDense` – this calls either `DGETRS` or `SGETRS` to use the LU factors and `pivots` array to perform the solve.
- `SUNLinSolLastFlag_LapackDense`
- `SUNLinSolSpace_LapackDense` – this only returns information for the storage *within* the solver object, i.e. storage for `N`, `last_flag`, and `pivots`.
- `SUNLinSolFree_LapackDense`

The module SUNLINSOL_LAPACKDENSE provides the following additional user-callable constructor routine:

- `SUNLapackDense`

  This function creates and allocates memory for a LAPACK dense `SUNLinearSolver`. Its arguments are an NVECTOR and SUNMATRIX, that it uses to determine the linear system size and to assess compatibility with the linear solver implementation.

  This routine will perform consistency checks to ensure that it is called with consistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX_DENSE matrix type and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

  If either `A` or `y` are incompatible then this routine will return `NULL`.

  ```
  SUNLinearSolver SUNLapackDense(N_Vector y, SUNMatrix A);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_LAPACKDENSE module also includes the Fortran-callable function `FSUNLapackDenseInit(code, ier)` to initialize this SUNLINSOL_LAPACKDENSE module for a given SUNDIALS solver. Here `code` is an integer input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); `ier` is an error return flag equal to 0 for success and -1 for failure. Both `code` and `ier` are declared to match C type `int`. This routine must be called *after* both the NVECTOR and SUNMATRIX objects have been initialized. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function `FSUNMassLapackDenseInit(ier)` initializes this SUNLINSOL_LAPACKDENSE module for solving mass matrix linear systems.

8.6 The SUNLinearSolver_LapackBand implementation

The LAPACK band implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_LAPACKBAND, is designed to be used with the corresponding SUNMATRIX_BAND matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, or NVECTOR_PTHREADS). The SUNLINSOL_LAPACKBAND module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_Band {
  sunindextype N;
  sunindextype *pivots;
  long int last_flag;
};
```

These entries of the *content* field contain the following information:

N - size of the linear system,

pivots - index array for partial pivoting in LU factorization,

last_flag - last error return flag from internal function evaluations.

The SUNLINSOL_LAPACKBAND module is a SUNLINSOL wrapper for the LAPACK band matrix factorization and solve routines, `*GBTRF` and `*GBTRS`, where `*` is either `D` or `S`, depending on whether SUNDIALS was configured to have `realtype` set to `double` or `single`, respectively (see Section 4.2). In order to use the SUNLINSOL_LAPACKBAND module it is assumed that LAPACK has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with LAPACK (see Appendix A for details). We note that since there do not exist 128-bit floating-point factorization and solve routines in LAPACK, this interface cannot be compiled when using `extended` precision for `realtype`. Similarly, since there do not exist 64-bit integer LAPACK routines, the SUNLINSOL_LAPACKBAND module also cannot be compiled when using `int64_t` for the `sunindextype`.

This solver is constructed to perform the following operations:

- The "setup" call performs a LU factorization with partial (row) pivoting, $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix with 1's on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX_BAND object A, with pivoting information encoding P stored in the `pivots` array.

- The "solve" call performs pivoting and forward and backward substitution using the stored `pivots` array and the LU factors held in the SUNMATRIX_BAND object.

- A must be allocated to accommodate the increase in upper bandwidth that occurs during factorization. More precisely, if A is a band matrix with upper bandwidth `mu` and lower bandwidth `ml`, then the upper triangular factor U can have upper bandwidth as big as `smu = MIN(N-1,mu+ml)`. The lower triangular factor L has lower bandwidth `ml`.

The header file to include when using this module is `sunlinsol/sunlinsol_lapackband.h`. The installed module library to link to is `libsundials_sunlinsollapackband.`*lib* where `.`*lib* is typically `.so` for shared libraries and `.a` for static libraries.
The SUNLINSOL_LAPACKBAND module defines band implementations of all "direct" linear solver operations listed in Table 8.2:

- `SUNLinSolGetType_LapackBand`
- `SUNLinSolInitialize_LapackBand` – this does nothing, since all consistency checks are performed at solver creation.
- `SUNLinSolSetup_LapackBand` – this calls either `DGBTRF` or `SGBTRF` to perform the *LU* factorization.
- `SUNLinSolSolve_LapackBand` – this calls either `DGBTRS` or `SGBTRS` to use the *LU* factors and `pivots` array to perform the solve.
- `SUNLinSolLastFlag_LapackBand`
- `SUNLinSolSpace_LapackBand` – this only returns information for the storage *within* the solver object, i.e. storage for `N`, `last_flag`, and `pivots`.
- `SUNLinSolFree_LapackBand`

The module SUNLINSOL_LAPACKBAND provides the following additional user-callable routine:

- `SUNLapackBand`

  This function creates and allocates memory for a LAPACK band `SUNLinearSolver`. Its arguments are an NVECTOR and SUNMATRIX, that it uses to determine the linear system size and to assess compatibility with the linear solver implementation.

  This routine will perform consistency checks to ensure that it is called with consistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX_BAND matrix type and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

  Additionally, this routine will verify that the input matrix `A` is allocated with appropriate upper bandwidth storage for the *LU* factorization.

  If either `A` or `y` are incompatible then this routine will return `NULL`.

  ```
  SUNLinearSolver SUNLapackBand(N_Vector y, SUNMatrix A);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_LAPACKBAND module also includes the Fortran-callable function `FSUNLapackBandInit(code, ier)` to initialize this SUNLINSOL_LAPACKBAND module for a given SUNDIALS solver. Here `code` is an integer input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); `ier` is an error return flag equal to 0 for success and -1 for failure. Both `code` and `ier` are declared to match C type `int`. This routine must be called *after* both the NVECTOR and SUNMATRIX objects have been initialized. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function `FSUNMassLapackBandInit(ier)` initializes this SUNLINSOL_LAPACKBAND module for solving mass matrix linear systems.

8.7 The SUNLinearSolver_KLU implementation

The KLU implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_KLU, is designed to be used with the corresponding SUNMATRIX_SPARSE matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, or NVECTOR_PTHREADS). The SUNLINSOL_KLU module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_KLU {
  long int         last_flag;
  int              first_factorize;
  sun_klu_symbolic *symbolic;
  sun_klu_numeric  *numeric;
  sun_klu_common   common;
  sunindextype     (*klu_solver)(sun_klu_symbolic*, sun_klu_numeric*,
                                 sunindextype, sunindextype,
                                 double*, sun_klu_common*);
};
```

These entries of the *content* field contain the following information:

last_flag - last error return flag from internal function evaluations,

first_factorize - flag indicating whether the factorization has ever been performed,

Symbolic - KLU storage structure for symbolic factorization components,

Numeric - KLU storage structure for numeric factorization components,

Common - storage structure for common KLU solver components,

klu_solver – pointer to the appropriate KLU solver function (depending on whether it is using a CSR or CSC sparse matrix).

The SUNLINSOL_KLU module is a SUNLINSOL wrapper for the KLU sparse matrix factorization and solver library written by Tim Davis [1, 9]. In order to use the SUNLINSOL_KLU interface to KLU, it is assumed that KLU has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with KLU (see Appendix A for details). Additionally, this wrapper only supports double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to have `realtype` set to either `extended` or `single` (see Section 4.2). Since the KLU library supports both 32-bit and 64-bit integers, this interface will be compiled for either of the available `sunindextype` options.

The KLU library has a symbolic factorization routine that computes the permutation of the linear system matrix to block triangular form and the permutations that will pre-order the diagonal blocks (the only ones that need to be factored) to reduce fill-in (using AMD, COLAMD, CHOLAMD, natural, or an ordering given by the user). Of these ordering choices, the default value in the SUNLINSOL_KLU module is the COLAMD ordering.

KLU breaks the factorization into two separate parts. The first is a symbolic factorization and the second is a numeric factorization that returns the factored matrix along with final pivot information. KLU also has a refactor routine that can be called instead of the numeric factorization. This routine will reuse the pivot information. This routine also returns diagnostic information that a user can examine to determine if numerical stability is being lost and a full numerical factorization should be done instead of the refactor.

Since the linear systems that arise within the context of SUNDIALS calculations will typically have identical sparsity patterns, the SUNLINSOL_KLU module is constructed to perform the following operations:

- The first time that the "setup" routine is called, it performs the symbolic factorization, followed by an initial numerical factorization.
- On subsequent calls to the "setup" routine, it calls the appropriate KLU "refactor" routine, followed by estimates of the numerical conditioning using the relevant "rcond", and if necessary "condest", routine(s). If these estimates of the condition number are larger than $\varepsilon^{-2/3}$ (where ε is the double-precision unit roundoff), then a new factorization is performed.
- The module includes the routine `SUNKLUReInit`, that can be called by the user to force a full refactorization at the next "setup" call.

- The "solve" call performs pivoting and forward and backward substitution using the stored KLU data structures. We note that in this solve KLU operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

The header file to include when using this module is `sunlinsol/sunlinsol_klu.h`. The installed module library to link to is `libsundials_sunlinsolklu.`*`lib`* where *`.lib`* is typically `.so` for shared libraries and `.a` for static libraries.
The SUNLINSOL_KLU module defines implementations of all "direct" linear solver operations listed in Table 8.2:

- `SUNLinSolGetType_KLU`
- `SUNLinSolInitialize_KLU` – this sets the `first_factorize` flag to 1, forcing both symbolic and numerical factorizations on the subsequent "setup" call.
- `SUNLinSolSetup_KLU` – this performs either a *LU* factorization or refactorization of the input matrix.
- `SUNLinSolSolve_KLU` – this calls the appropriate KLU solve routine to utilize the *LU* factors to solve the linear system.
- `SUNLinSolLastFlag_KLU`
- `SUNLinSolSpace_KLU` – this only returns information for the storage within the solver *interface*, i.e. storage for the integers `last_flag` and `first_factorize`. For additional space requirements, see the KLU documentation.
- `SUNLinSolFree_KLU`

The module SUNLINSOL_KLU provides the following additional user-callable routines:

- `SUNKLU`

  This constructor function creates and allocates memory for a SUNLINSOL_KLU object. Its arguments are an NVECTOR and SUNMATRIX, that it uses to determine the linear system size and to assess compatibility with the linear solver implementation.

  This routine will perform consistency checks to ensure that it is called with consistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX_SPARSE matrix type (using either CSR or CSC storage formats) and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

  If either `A` or `y` are incompatible then this routine will return `NULL`.

  ```
  SUNLinearSolver SUNKLU(N_Vector y, SUNMatrix A);
  ```

- `SUNKLUReInit`

  This function reinitializes memory and flags for a new factorization (symbolic and numeric) to be conducted at the next solver setup call. This routine is useful in the cases where the number of nonzeroes has changed or if the structure of the linear system has changed which would require a new symbolic (and numeric factorization).

  The `reinit_type` argument governs the level of reinitialization. The allowed values are:

  1 The Jacobian matrix will be destroyed and a new one will be allocated based on the `nnz` value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup.

  2 Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of `nnz` given in the sparse matrix provided to the original constructor routine (or the previous `SUNKLUReInit` call).

This routine assumes no other changes to solver use are necessary.

The return values from this function are SUNLS_MEM_NULL (either S or A are NULL), SUNLS_ILL_INPUT (A does not have type SUNMATRIX_SPARSE or reinit_type is invalid), SUNLS_MEM_FAIL (reallocation of the sparse matrix failed) or SUNLS_SUCCESS.

```
int SUNKLUReInit(SUNLinearSolver S, SUNMatrix A,
                 sunindextype nnz, int reinit_type);
```

- SUNKLUSetOrdering

  This function sets the ordering used by KLU for reducing fill in the linear solve. Options for ordering_choice are:

  0 AMD,

  1 COLAMD, and

  2 the natural ordering.

  The default is 1 for COLAMD.

  The return values from this function are SUNLS_MEM_NULL (S is NULL), SUNLS_ILL_INPUT (invalid ordering_choice), or SUNLS_SUCCESS.

  ```
  int SUNKLUSetOrdering(SUNLinearSolver S, int ordering_choice);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_KLU module also includes the Fortran-callable function FSUNKLUInit(code, ier) to initialize this SUNLINSOL_KLU module for a given SUNDIALS solver. Here code is an integer input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); ier is an error return flag equal to 0 for success and -1 for failure. Both code and ier are declared to match C type int. This routine must be called *after* both the NVECTOR and SUNMATRIX objects have been initialized. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function FSUNMassKLUInit(ier) initializes this SUNLINSOL_KLU module for solving mass matrix linear systems.

The SUNKLUReInit and SUNKLUSetOrdering routines also support Fortran interfaces for the system and mass matrix solvers:

- FSUNKLUReInit(code, NNZ, reinit_type, ier) – NNZ should be commensurate with a C long int and reinit_type should be commensurate with a C int
- FSUNMassKLUReInit(NNZ, reinit_type, ier)
- FSUNKLUSetOrdering(code, ordering, ier) – ordering should be commensurate with a C int
- FSUNMassKLUSetOrdering(ordering, ier)

8.8 The SUNLinearSolver_SuperLUMT implementation

The SUPERLUMT implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_SUPERLUMT, is designed to be used with the corresponding SUNMATRIX_SPARSE matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, or NVECTOR_PTHREADS). While these are compatible, it is not recommended to use a threaded vector module with SUNLINSOL_SUPERLUMT unless it is the NVECTOR_OPENMP module and the SUPERLUMT library has also been compiled with OpenMP. The SUNLINSOL_SUPERLUMT module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_SuperLUMT {
  long int     last_flag;
  int          first_factorize;
  SuperMatrix  *A, *AC, *L, *U, *B;
  Gstat_t      *Gstat;
  sunindextype *perm_r, *perm_c;
  sunindextype N;
  int          num_threads;
  realtype     diag_pivot_thresh;
  int          ordering;
  superlumt_options_t *options;
};
```

These entries of the *content* field contain the following information:

last_flag - last error return flag from internal function evaluations,

first_factorize - flag indicating whether the factorization has ever been performed,

A, AC, L, U, B - `SuperMatrix` pointers used in solve,

Gstat - `GStat_t` object used in solve,

perm_r, perm_c - permutation arrays used in solve,

N - size of the linear system,

num_threads - number of OpenMP/Pthreads threads to use,

diag_pivot_thresh - threshold on diagonal pivoting,

ordering - flag for which reordering algorithm to use,

options - pointer to SUPERLUMT options structure.

The SUNLINSOL_SUPERLUMT module is a SUNLINSOL wrapper for the SUPERLUMT sparse matrix factorization and solver library written by X. Sherry Li [2, 21, 11]. The package performs matrix factorization using threads to enhance efficiency in shared memory parallel environments. It should be noted that threads are only used in the factorization step. In order to use the SUNLINSOL_SUPERLUMT interface to SUPERLUMT, it is assumed that SUPERLUMT has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with SUPERLUMT (see Appendix A for details). Additionally, this wrapper only supports single- and double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to have `realtype` set to `extended` (see Section 4.2). Moreover, since the SUPERLUMT library may be installed to support either 32-bit or 64-bit integers, it is assumed that the SUPERLUMT library is installed using the same integer precision as the SUNDIALS `sunindextype` option.

The SUPERLUMT library has a symbolic factorization routine that computes the permutation of the linear system matrix to reduce fill-in on subsequent *LU* factorizations (using COLAMD, minimal degree ordering on $A^T * A$, minimal degree ordering on $A^T + A$, or natural ordering). Of these ordering choices, the default value in the SUNLINSOL_SUPERLUMT module is the COLAMD ordering.

Since the linear systems that arise within the context of SUNDIALS calculations will typically have identical sparsity patterns, the SUNLINSOL_SUPERLUMT module is constructed to perform the following operations:

- The first time that the "setup" routine is called, it performs the symbolic factorization, followed by an initial numerical factorization.
- On subsequent calls to the "setup" routine, it skips the symbolic factorization, and only refactors the input matrix.

- The "solve" call performs pivoting and forward and backward substitution using the stored SUPERLUMT data structures. We note that in this solve SUPERLUMT operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

The header file to include when using this module is `sunlinsol/sunlinsol_superlumt.h`. The installed module library to link to is `libsundials_sunlinsolsuperlumt.`*`lib`* where *`.lib`* is typically `.so` for shared libraries and `.a` for static libraries.
The SUNLINSOL_SUPERLUMT module defines implementations of all "direct" linear solver operations listed in Table 8.2:

- `SUNLinSolGetType_SuperLUMT`
- `SUNLinSolInitialize_SuperLUMT` – this sets the `first_factorize` flag to 1 and resets the internal SUPERLUMT statistics variables.
- `SUNLinSolSetup_SuperLUMT` – this performs either a *LU* factorization or refactorization of the input matrix.
- `SUNLinSolSolve_SuperLUMT` – this calls the appropriate SUPERLUMT solve routine to utilize the *LU* factors to solve the linear system.
- `SUNLinSolLastFlag_SuperLUMT`
- `SUNLinSolSpace_SuperLUMT` – this only returns information for the storage within the solver *interface*, i.e. storage for the integers `last_flag` and `first_factorize`. For additional space requirements, see the SUPERLUMT documentation.
- `SUNLinSolFree_SuperLUMT`

The module SUNLINSOL_SUPERLUMT provides the following additional user-callable routines:

- `SUNSuperLUMT`

  This constructor function creates and allocates memory for a SUNLINSOL_SUPERLUMT object. Its arguments are an NVECTOR, a SUNMATRIX, and a desired number of threads (OpenMP or Pthreads, depending on how SUPERLUMT was installed) to use during the factorization steps. This routine analyzes the input matrix and vector to determine the linear system size and to assess compatibility with the SUPERLUMT library.

  This routine will perform consistency checks to ensure that it is called with consistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX_SPARSE matrix type (using either CSR or CSC storage formats) and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

  If either `A` or `y` are incompatible then this routine will return `NULL`. The `num_threads` argument is not checked and is passed directly to SUPERLUMT routines.

  ```
  SUNLinearSolver SUNSuperLUMT(N_Vector y, SUNMatrix A, int num_threads);
  ```

- `SUNSuperLUMTSetOrdering`

  This function sets the ordering used by SUPERLUMT for reducing fill in the linear solve. Options for `ordering_choice` are:

  0 natural ordering

  1 minimal degree ordering on $A^T A$

  2 minimal degree ordering on $A^T + A$

  3 COLAMD ordering for unsymmetric matrices

The default is 3 for COLAMD.

The return values from this function are `SUNLS_MEM_NULL` (`S` is `NULL`), `SUNLS_ILL_INPUT` (invalid `ordering_choice`), or `SUNLS_SUCCESS`.

```
int SUNSuperLUMTSetOrdering(SUNLinearSolver S, int ordering_choice);
```

For solvers that include a Fortran interface module, the SUNLINSOL_SUPERLUMT module also includes the Fortran-callable function `FSUNSuperLUMTInit(code, num_threads, ier)` to initialize this SUNLINSOL_SUPERLUMT module for a given SUNDIALS solver. Here `code` is an integer input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); `num_threads` is the desired number of OpenMP/Pthreads threads to use in the factorization; `ier` is an error return flag equal to 0 for success and -1 for failure. All of these arguments should be declared so as to match C type `int`. This routine must be called *after* both the NVECTOR and SUNMATRIX objects have been initialized. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function `FSUNMassSuperLUMTInit(num_threads, ier)` initializes this SUNLINSOL_SUPERLUMT module for solving mass matrix linear systems.

The `SUNSuperLUMTSetOrdering` routine also supports Fortran interfaces for the system and mass matrix solvers:

- `FSUNSuperLUMTSetOrdering(code, ordering, ier)` – `ordering` should be commensurate with a C `int`
- `FSUNMassSuperLUMTSetOrdering(ordering, ier)`

8.9 The SUNLinearSolver_SPGMR implementation

The SPGMR (Scaled, Preconditioned, Generalized Minimum Residual [26]) implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_SPGMR, is an iterative linear solver that is designed to be compatible with any NVECTOR implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (`N_VClone`, `N_VDotProd`, `N_VScale`, `N_VLinearSum`, `N_VProd`, `N_VConst`, `N_VDiv`, and `N_VDestroy`).

The SUNLINSOL_SPGMR module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_SPGMR {
  int maxl;
  int pretype;
  int gstype;
  int max_restarts;
  int numiters;
  realtype resnorm;
  long int last_flag;
  ATimesFn ATimes;
  void* ATData;
  PSetupFn Psetup;
  PSolveFn Psolve;
  void* PData;
  N_Vector s1;
  N_Vector s2;
  N_Vector *V;
  realtype **Hes;
  realtype *givens;
  N_Vector xcor;
  realtype *yg;
  N_Vector vtemp;
};
```

These entries of the *content* field contain the following information:

maxl - number of GMRES basis vectors to use (default is 5),

pretype - flag for type of preconditioning to employ (default is none),

gstype - flag for type of Gram-Schmidt orthogonalization (default is modified Gram-Schmidt),

max_restarts - number of GMRES restarts to allow (default is 0),

numiters - number of iterations from the most-recent solve,

resnorm - final linear residual norm from the most-recent solve,

last_flag - last error return flag from an internal function,

ATimes - function pointer to perform Av product,

ATData - pointer to structure for `ATimes`,

Psetup - function pointer to preconditioner setup routine,

Psolve - function pointer to preconditioner solve routine,

PData - pointer to structure for `Psetup` and `Psolve`,

s1, s2 - vector pointers for supplied scaling matrices (default is `NULL`),

V - the array of Krylov basis vectors $v_1, \ldots, v_{\texttt{maxl}+1}$, stored in `V[0]`, ..., `V[maxl]`. Each v_i is a vector of type NVECTOR.,

Hes - the $(\texttt{maxl}+1) \times \texttt{maxl}$ Hessenberg matrix. It is stored row-wise so that the (i,j)th element is given by `Hes[i][j]`.,

givens - a length `2*maxl` array which represents the Givens rotation matrices that arise in the GMRES algorithm. These matrices are $F_0, F_1, \ldots, F_j$, where $F_i = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & c_i & -s_i & & \\ & & & s_i & c_i & & \\ & & & & & 1 & \\ & & & & & & \ddots \\ & & & & & & & 1 \end{bmatrix}$, are represented in the `givens` vector as `givens[0]` $= c_0$, `givens[1]` $= s_0$, `givens[2]` $= c_1$, `givens[3]` $= s_1$, ... `givens[2j]` $= c_j$, `givens[2j+1]` $= s_j$.,

xcor - a vector which holds the scaled, preconditioned correction to the initial guess,

yg - a length `(maxl+1)` array of `realtype` values used to hold "short" vectors (e.g. y and g),

vtemp - temporary vector storage.

This solver is constructed to perform the following operations:

- During construction, the `xcor` and `vtemp` arrays are cloned from a template NVECTOR that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLINSOL_SPGMR to supply the `ATimes`, `PSetup`, and `Psolve` function pointers and `s1` and `s2` scaling vectors.

- In the "initialize" call, the remaining solver data is allocated (`V`, `Hes`, `givens`, and `yg`)
- In the "setup" call, any non-`NULL` `PSetup` function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call, the GMRES iteration is performed. This will include scaling, preconditioning, and restarts if those options have been supplied.

The header file to include when using this module is `sunlinsol/sunlinsol_spgmr.h`. The SUNLINSOL_SPGMR module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolspgmr` module library.
The SUNLINSOL_SPGMR module defines implementations of all "iterative" linear solver operations listed in Table 8.2:

- `SUNLinSolGetType_SPGMR`
- `SUNLinSolInitialize_SPGMR`
- `SUNLinSolSetATimes_SPGMR`
- `SUNLinSolSetPreconditioner_SPGMR`
- `SUNLinSolSetScalingVectors_SPGMR`
- `SUNLinSolSetup_SPGMR`
- `SUNLinSolSolve_SPGMR`
- `SUNLinSolNumIters_SPGMR`
- `SUNLinSolResNorm_SPGMR`
- `SUNLinSolResid_SPGMR`
- `SUNLinSolLastFlag_SPGMR`
- `SUNLinSolSpace_SPGMR`
- `SUNLinSolFree_SPGMR`

The module SUNLINSOL_SPGMR provides the following additional user-callable routines:

- `SUNSPGMR`

  This constructor function creates and allocates memory for a SPGMR `SUNLinearSolver`. Its arguments are an NVECTOR, the desired type of preconditioning, and the number of Krylov basis vectors to use.

  This routine will perform consistency checks to ensure that it is called with a consistent NVECTOR implementation (i.e. that it supplies the requisite vector operations). If `y` is incompatible, then this routine will return `NULL`.

  A `maxl` argument that is ≤ 0 will result in the default value (5).

  Allowable inputs for `pretype` are `PREC_NONE` (0), `PREC_LEFT` (1), `PREC_RIGHT` (2) and `PREC_BOTH` (3); any other integer input will result in the default (no preconditioning). We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL). While it is possible to configure a SUNLINSOL_SPGMR object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.

  ```
  SUNLinearSolver SUNSPGMR(N_Vector y, int pretype, int maxl);
  ```

- SUNSPGMRSetPrecType

  This function updates the type of preconditioning to use. Supported values are PREC_NONE (0), PREC_LEFT (1), PREC_RIGHT (2) and PREC_BOTH (3).

  This routine will return with one of the error codes SUNLS_ILL_INPUT (illegal pretype), SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

  ```
  int SUNSPGMRSetPrecType(SUNLinearSolver S, int pretype);
  ```

- SUNSPGMRSetGSType

  This function sets the type of Gram-Schmidt orthogonalization to use. Supported values are MODIFIED_GS (1) and CLASSICAL_GS (2). Any other integer input will result in a failure, returning error code SUNLS_ILL_INPUT.

  This routine will return with one of the error codes SUNLS_ILL_INPUT (illegal gstype), SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

  ```
  int SUNSPGMRSetGSType(SUNLinearSolver S, int gstype);
  ```

- SUNSPGMRSetMaxRestarts

  This function sets the number of GMRES restarts to allow. A negative input will result in the default of 0.

  This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

  ```
  int SUNSPGMRSetMaxRestarts(SUNLinearSolver S, int maxrs);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_SPGMR module also includes the Fortran-callable function FSUNSPGMRInit(code, pretype, maxl, ier) to initialize this SUNLINSOL_SPGMR module for a given SUNDIALS solver. Here code is an integer input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); pretype and maxl are the same as for the C function SUNSPGMR; ier is an error return flag equal to 0 for success and -1 for failure. All of these input arguments should be declared so as to match C type int. This routine must be called *after* the NVECTOR object has been initialized. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function FSUNMassSPGMRInit(pretype, maxl, ier) initializes this SUNLINSOL_SPGMR module for solving mass matrix linear systems.

The SUNSPGMRSetPrecType, SUNSPGMRSetGSType and SUNSPGMRSetMaxRestarts routines also support Fortran interfaces for the system and mass matrix solvers (all arguments should be commensurate with a C int):

- FSUNSPGMRSetGSType(code, gstype, ier)
- FSUNMassSPGMRSetGSType(gstype, ier)
- FSUNSPGMRSetPrecType(code, pretype, ier)
- FSUNMassSPGMRSetPrecType(pretype, ier)
- FSUNSPGMRSetMaxRS(code, maxrs, ier)
- FSUNMassSPGMRSetMaxRS(maxrs, ier)

8.10 The SUNLinearSolver_SPFGMR implementation

The SPFGMR (Scaled, Preconditioned, Flexible, Generalized Minimum Residual [25]) implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_SPFGMR, is an iterative linear solver that is designed to be compatible with any NVECTOR implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (N_VClone, N_VDotProd, N_VScale, N_VLinearSum, N_VProd, N_VConst, N_VDiv, and N_VDestroy). Unlike the other Krylov iterative linear

solvers supplied with SUNDIALS, FGMRES is specifically designed to work with a changing preconditioner (e.g. from an iterative method).

The SUNLINSOL_SPFGMR module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_SPFGMR {
  int maxl;
  int pretype;
  int gstype;
  int max_restarts;
  int numiters;
  realtype resnorm;
  long int last_flag;
  ATimesFn ATimes;
  void* ATData;
  PSetupFn Psetup;
  PSolveFn Psolve;
  void* PData;
  N_Vector s1;
  N_Vector s2;
  N_Vector *V;
  N_Vector *Z;
  realtype **Hes;
  realtype *givens;
  N_Vector xcor;
  realtype *yg;
  N_Vector vtemp;
};
```

These entries of the *content* field contain the following information:

maxl - number of FGMRES basis vectors to use (default is 5),

pretype - flag for use of preconditioning (default is none),

gstype - flag for type of Gram-Schmidt orthogonalization (default is modified Gram-Schmidt),

max_restarts - number of FGMRES restarts to allow (default is 0),

numiters - number of iterations from the most-recent solve,

resnorm - final linear residual norm from the most-recent solve,

last_flag - last error return flag from an internal function,

ATimes - function pointer to perform Av product,

ATData - pointer to structure for `ATimes`,

Psetup - function pointer to preconditioner setup routine,

Psolve - function pointer to preconditioner solve routine,

PData - pointer to structure for `Psetup` and `Psolve`,

s1, s2 - vector pointers for supplied scaling matrices (default is `NULL`),

V - the array of Krylov basis vectors $v_1, \ldots, v_{\texttt{maxl}+1}$, stored in `V[0]`, ..., `V[maxl]`. Each v_i is a vector of type NVECTOR.,

Z - the array of preconditioned Krylov basis vectors $z_1, \ldots, z_{\texttt{maxl}+1}$, stored in `Z[0]`, ..., `Z[maxl]`. Each z_i is a vector of type NVECTOR.,

Hes - the $(\texttt{maxl}+1) \times \texttt{maxl}$ Hessenberg matrix. It is stored row-wise so that the (i,j)th element is given by `Hes[i][j]`.,

givens - a length `2*maxl` array which represents the Givens rotation matrices that arise in the FGMRES algorithm. These matrices are $F_0, F_1, \ldots, F_j$, where $F_i = \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_i & -s_i & & & \\ & & & s_i & c_i & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix}$, are represented in the `givens` vector as `givens[0]` $= c_0$, `givens[1]` $= s_0$, `givens[2]` $= c_1$, `givens[3]` $= s_1$, ... `givens[2j]` $= c_j$, `givens[2j+1]` $= s_j$.,

xcor - a vector which holds the scaled, preconditioned correction to the initial guess,

yg - a length `(maxl+1)` array of `realtype` values used to hold "short" vectors (e.g. y and g),

vtemp - temporary vector storage.

This solver is constructed to perform the following operations:

- During construction, the `xcor` and `vtemp` arrays are cloned from a template NVECTOR that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLINSOL_SPFGMR to supply the `ATimes`, `PSetup`, and `Psolve` function pointers and `s1` and `s2` scaling vectors.
- In the "initialize" call, the remaining solver data is allocated (`V`, `Hes`, `givens`, and `yg`)
- In the "setup" call, any non-`NULL` `PSetup` function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call, the FGMRES iteration is performed. This will include scaling, preconditioning, and restarts if those options have been supplied.

The header file to include when using this module is `sunlinsol/sunlinsol_spfgmr.h`. The SUNLINSOL_SPFGMR module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolspfgmr` module library.

The SUNLINSOL_SPFGMR module defines implementations of all "iterative" linear solver operations listed in Table 8.2:

- `SUNLinSolGetType_SPFGMR`
- `SUNLinSolInitialize_SPFGMR`
- `SUNLinSolSetATimes_SPFGMR`
- `SUNLinSolSetPreconditioner_SPFGMR`
- `SUNLinSolSetScalingVectors_SPFGMR`
- `SUNLinSolSetup_SPFGMR`

- SUNLinSolSolve_SPFGMR
- SUNLinSolNumIters_SPFGMR
- SUNLinSolResNorm_SPFGMR
- SUNLinSolResid_SPFGMR
- SUNLinSolLastFlag_SPFGMR
- SUNLinSolSpace_SPFGMR
- SUNLinSolFree_SPFGMR

The module SUNLINSOL_SPFGMR provides the following additional user-callable routines:

- SUNSPFGMR

  This constructor function creates and allocates memory for a SPFGMR SUNLinearSolver. Its arguments are an NVECTOR, a flag indicating to use preconditioning, and the number of Krylov basis vectors to use.

  This routine will perform consistency checks to ensure that it is called with a consistent NVECTOR implementation (i.e. that it supplies the requisite vector operations). If y is incompatible, then this routine will return NULL.

  A maxl argument that is ≤ 0 will result in the default value (5).

  Since the FGMRES algorithm is designed to only support right preconditioning, then any of the pretype inputs PREC_LEFT (1), PREC_RIGHT (2), or PREC_BOTH (3) will result in use of PREC_RIGHT; any other integer input will result in the default (no preconditioning). We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS). While it is possible to use a right-preconditioned SUNLINSOL_SPFGMR object for these packages, this use mode is not supported and may result in inferior performance.

  ```
  SUNLinearSolver SUNSPFGMR(N_Vector y, int pretype, int maxl);
  ```

- SUNSPFGMRSetPrecType

  This function updates the flag indicating use of preconditioning. Since the FGMRES algorithm is designed to only support right preconditioning, then any of the pretype inputs PREC_LEFT (1), PREC_RIGHT (2), or PREC_BOTH (3) will result in use of PREC_RIGHT; any other integer input will result in the default (no preconditioning).

  This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

  ```
  int SUNSPFGMRSetPrecType(SUNLinearSolver S, int pretype);
  ```

- SUNSPFGMRSetGSType

  This function sets the type of Gram-Schmidt orthogonalization to use. Supported values are MODIFIED_GS (1) and CLASSICAL_GS (2). Any other integer input will result in a failure, returning error code SUNLS_ILL_INPUT.

  This routine will return with one of the error codes SUNLS_ILL_INPUT (illegal gstype), SUNLS_MEM_NULL (S is NULL), or SUNLS_SUCCESS.

  ```
  int SUNSPFGMRSetGSType(SUNLinearSolver S, int gstype);
  ```

- SUNSPFGMRSetMaxRestarts

  This function sets the number of FGMRES restarts to allow. A negative input will result in the default of 0.

  This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

  ```
  int SUNSPFGMRSetMaxRestarts(SUNLinearSolver S, int maxrs);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_SPFGMR module also includes the Fortran-callable function `FSUNSPFGMRInit(code, pretype, maxl, ier)` to initialize this SUNLINSOL_SPFGMR module for a given SUNDIALS solver. Here `code` is an integer input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); `pretype` and `maxl` are the same as for the C function `SUNSPFGMR`; `ier` is an error return flag equal to 0 for success and -1 for failure. All of these input arguments should be declared so as to match C type `int`. This routine must be called *after* the NVECTOR object has been initialized. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function `FSUNMassSPFGMRInit(pretype, maxl, ier)` initializes this SUNLINSOL_SPFGMR module for solving mass matrix linear systems.

The `SUNSPFGMRSetPrecType`, `SUNSPFGMRSetGSType`, and `SUNSPFGMRSetMaxRestarts` routines also support Fortran interfaces for the system and mass matrix solvers (all arguments should be commensurate with a C `int`):

- `FSUNSPFGMRSetGSType(code, gstype, ier)`
- `FSUNMassSPFGMRSetGSType(gstype, ier)`
- `FSUNSPFGMRSetPrecType(code, pretype, ier)`
- `FSUNMassSPFGMRSetPrecType(pretype, ier)`
- `FSUNSPFGMRSetMaxRS(code, maxrs, ier)`
- `FSUNMassSPFGMRSetMaxRS(maxrs, ier)`

8.11 The SUNLinearSolver_SPBCGS implementation

The SPBCGS (Scaled, Preconditioned, Bi-Conjugate Gradient, Stabilized [27]) implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_SPBCGS, is an iterative linear solver that is designed to be compatible with any NVECTOR implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (`N_VClone`, `N_VDotProd`, `N_VScale`, `N_VLinearSum`, `N_VProd`, `N_VDiv`, and `N_VDestroy`). Unlike the SPGMR and SPFGMR algorithms, SPBCGS requires a fixed amount of memory that does not increase with the number of allowed iterations.

The SUNLINSOL_SPBCGS module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_SPBCGS {
  int maxl;
  int pretype;
  int numiters;
  realtype resnorm;
  long int last_flag;
  ATimesFn ATimes;
  void* ATData;
  PSetupFn Psetup;
  PSolveFn Psolve;
  void* PData;
  N_Vector s1;
  N_Vector s2;
  N_Vector r;
  N_Vector r_star;
  N_Vector p;
  N_Vector q;
  N_Vector u;
  N_Vector Ap;
  N_Vector vtemp;
};
```

These entries of the *content* field contain the following information:

maxl - number of SPBCGS iterations to allow (default is 5),

pretype - flag for type of preconditioning to employ (default is none),

numiters - number of iterations from the most-recent solve,

resnorm - final linear residual norm from the most-recent solve,

last_flag - last error return flag from an internal function,

ATimes - function pointer to perform Av product,

ATData - pointer to structure for `ATimes`,

Psetup - function pointer to preconditioner setup routine,

Psolve - function pointer to preconditioner solve routine,

PData - pointer to structure for `Psetup` and `Psolve`,

s1, s2 - vector pointers for supplied scaling matrices (default is `NULL`),

r - a NVECTOR which holds the current scaled, preconditioned linear system residual,

r_star - a NVECTOR which holds the initial scaled, preconditioned linear system residual,

p, q, u, Ap, vtemp - NVECTORs used for workspace by the SPBCGS algorithm.

This solver is constructed to perform the following operations:

- During construction all NVECTOR solver data is allocated, with vectors cloned from a template NVECTOR that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLINSOL_SPBCGS to supply the `ATimes`, `PSetup`, and `Psolve` function pointers and `s1` and `s2` scaling vectors.
- In the "initialize" call, the solver parameters are checked for validity.
- In the "setup" call, any non-`NULL` `PSetup` function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call the SPBCGS iteration is performed. This will include scaling and preconditioning if those options have been supplied.

The header file to include when using this module is `sunlinsol/sunlinsol_spbcgs.h`. The SUNLINSOL_SPBCGS module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolspbcgs` module library.

The SUNLINSOL_SPBCGS module defines implementations of all "iterative" linear solver operations listed in Table 8.2:

- `SUNLinSolGetType_SPBCGS`
- `SUNLinSolInitialize_SPBCGS`
- `SUNLinSolSetATimes_SPBCGS`
- `SUNLinSolSetPreconditioner_SPBCGS`
- `SUNLinSolSetScalingVectors_SPBCGS`

- SUNLinSolSetup_SPBCGS
- SUNLinSolSolve_SPBCGS
- SUNLinSolNumIters_SPBCGS
- SUNLinSolResNorm_SPBCGS
- SUNLinSolResid_SPBCGS
- SUNLinSolLastFlag_SPBCGS
- SUNLinSolSpace_SPBCGS
- SUNLinSolFree_SPBCGS

The module SUNLINSOL_SPBCGS provides the following additional user-callable routines:

- SUNSPBCGS

  This constructor function creates and allocates memory for a SPBCGS SUNLinearSolver. Its arguments are an NVECTOR, the desired type of preconditioning, and the number of linear iterations to allow.

  This routine will perform consistency checks to ensure that it is called with a consistent NVECTOR implementation (i.e. that it supplies the requisite vector operations). If y is incompatible, then this routine will return NULL.

  A maxl argument that is ≤ 0 will result in the default value (5).

  Allowable inputs for pretype are PREC_NONE (0), PREC_LEFT (1), PREC_RIGHT (2) and PREC_BOTH (3); any other integer input will result in the default (no preconditioning). We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL). While it is possible to configure a SUNLINSOL_SPBCGS object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.

  ```
  SUNLinearSolver SUNSPBCGS(N_Vector y, int pretype, int maxl);
  ```

- SUNSPBCGSSetPrecType

  This function updates the type of preconditioning to use. Supported values are PREC_NONE (0), PREC_LEFT (1), PREC_RIGHT (2), and PREC_BOTH (3).

  This routine will return with one of the error codes SUNLS_ILL_INPUT (illegal pretype), SUNLS_MEM_NULL (S is NULL), or SUNLS_SUCCESS.

  ```
  int SUNSPBCGSSetPrecType(SUNLinearSolver S, int pretype);
  ```

- SUNSPBCGSSetMaxl

  This function updates the number of linear solver iterations to allow.

  A maxl argument that is ≤ 0 will result in the default value (5).

  This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

  ```
  int SUNSPBCGSSetMaxl(SUNLinearSolver S, int maxl);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_SPBCGS module also includes the Fortran-callable function FSUNSPBCGSInit(code, pretype, maxl, ier) to initialize this SUNLINSOL_SPBCGS module for a given SUNDIALS solver. Here code is an integer input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); pretype and maxl are the same as for the C function SUNSPBCGS; ier is an error return flag equal to 0 for success and -1 for failure. All of these input arguments should be declared so as to match C type int. This routine must be called *after* the NVECTOR object has been initialized. Additionally, when using ARKODE with a non-identity

mass matrix, the Fortran-callable function `FSUNMassSPBCGSInit(pretype, maxl, ier)` initializes this SUNLINSOL_SPBCGS module for solving mass matrix linear systems.

The `SUNSPBCGSSetPrecType` and `SUNSPBCGSSetMaxl` routines also support Fortran interfaces for the system and mass matrix solvers (all arguments should be commensurate with a C `int`):

- `FSUNSPBCGSSetPrecType(code, pretype, ier)`
- `FSUNMassSPBCGSSetPrecType(pretype, ier)`
- `FSUNSPBCGSSetMaxl(code, maxl, ier)`
- `FSUNMassSPBCGSSetMaxl(maxl, ier)`

8.12 The SUNLinearSolver_SPTFQMR implementation

The SPTFQMR (Scaled, Preconditioned, Transpose-Free Quasi-Minimum Residual [15]) implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_SPTFQMR, is an iterative linear solver that is designed to be compatible with any NVECTOR implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (`N_VClone`, `N_VDotProd`, `N_VScale`, `N_VLinearSum`, `N_VProd`, `N_VConst`, `N_VDiv`, and `N_VDestroy`). Unlike the SPGMR and SPFGMR algorithms, SPTFQMR requires a fixed amount of memory that does not increase with the number of allowed iterations.

The SUNLINSOL_SPTFQMR module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_SPTFQMR {
  int maxl;
  int pretype;
  int numiters;
  realtype resnorm;
  long int last_flag;
  ATimesFn ATimes;
  void* ATData;
  PSetupFn Psetup;
  PSolveFn Psolve;
  void* PData;
  N_Vector s1;
  N_Vector s2;
  N_Vector r_star;
  N_Vector q;
  N_Vector d;
  N_Vector v;
  N_Vector p;
  N_Vector *r;
  N_Vector u;
  N_Vector vtemp1;
  N_Vector vtemp2;
  N_Vector vtemp3;
};
```

These entries of the *content* field contain the following information:

maxl - number of TFQMR iterations to allow (default is 5),

pretype - flag for type of preconditioning to employ (default is none),

numiters - number of iterations from the most-recent solve,

resnorm - final linear residual norm from the most-recent solve,

last_flag - last error return flag from an internal function,

ATimes - function pointer to perform Av product,

ATData - pointer to structure for `ATimes`,

Psetup - function pointer to preconditioner setup routine,

Psolve - function pointer to preconditioner solve routine,

PData - pointer to structure for `Psetup` and `Psolve`,

s1, s2 - vector pointers for supplied scaling matrices (default is `NULL`),

r_star - a NVECTOR which holds the initial scaled, preconditioned linear system residual,

q, d, v, p, u - NVECTORs used for workspace by the SPTFQMR algorithm,

r - array of two NVECTORs used for workspace within the SPTFQMR algorithm,

vtemp1, vtemp2, vtemp3 - temporary vector storage.

This solver is constructed to perform the following operations:

- During construction all NVECTOR solver data is allocated, with vectors cloned from a template NVECTOR that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLINSOL_SPTFQMR to supply the `ATimes`, `PSetup`, and `Psolve` function pointers and `s1` and `s2` scaling vectors.
- In the "initialize" call, the solver parameters are checked for validity.
- In the "setup" call, any non-`NULL` `PSetup` function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call the TFQMR iteration is performed. This will include scaling and preconditioning if those options have been supplied.

The header file to include when using this module is `sunlinsol/sunlinsol_sptfqmr.h`. The SUNLINSOL_SPTFQMR module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolsptfqmr` module library.

The SUNLINSOL_SPTFQMR module defines implementations of all "iterative" linear solver operations listed in Table 8.2:

- `SUNLinSolGetType_SPTFQMR`
- `SUNLinSolInitialize_SPTFQMR`
- `SUNLinSolSetATimes_SPTFQMR`
- `SUNLinSolSetPreconditioner_SPTFQMR`
- `SUNLinSolSetScalingVectors_SPTFQMR`
- `SUNLinSolSetup_SPTFQMR`
- `SUNLinSolSolve_SPTFQMR`
- `SUNLinSolNumIters_SPTFQMR`

- `SUNLinSolResNorm_SPTFQMR`
- `SUNLinSolResid_SPTFQMR`
- `SUNLinSolLastFlag_SPTFQMR`
- `SUNLinSolSpace_SPTFQMR`
- `SUNLinSolFree_SPTFQMR`

The module SUNLINSOL_SPTFQMR provides the following additional user-callable routines:

- `SUNSPTFQMR`

  This constructor function creates and allocates memory for a SPTFQMR `SUNLinearSolver`. Its arguments are an NVECTOR, the desired type of preconditioning, and the number of linear iterations to allow.

  This routine will perform consistency checks to ensure that it is called with a consistent NVECTOR implementation (i.e. that it supplies the requisite vector operations). If `y` is incompatible, then this routine will return `NULL`.

  A `maxl` argument that is ≤ 0 will result in the default value (5).

  Allowable inputs for `pretype` are `PREC_NONE` (0), `PREC_LEFT` (1), `PREC_RIGHT` (2) and `PREC_BOTH` (3); any other integer input will result in the default (no preconditioning). We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL). While it is possible to configure a SUNLINSOL_SPTFQMR object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.

  ```
  SUNLinearSolver SUNSPTFQMR(N_Vector y, int pretype, int maxl);
  ```

- `SUNSPTFQMRSetPrecType`

  This function updates the type of preconditioning to use. Supported values are `PREC_NONE` (0), `PREC_LEFT` (1), `PREC_RIGHT` (2), and `PREC_BOTH` (3).

  This routine will return with one of the error codes `SUNLS_ILL_INPUT` (illegal `pretype`), `SUNLS_MEM_NULL` (`S` is `NULL`), or `SUNLS_SUCCESS`.

  ```
  int SUNSPTFQMRSetPrecType(SUNLinearSolver S, int pretype);
  ```

- `SUNSPTFQMRSetMaxl`

  This function updates the number of linear solver iterations to allow.

  A `maxl` argument that is ≤ 0 will result in the default value (5).

  This routine will return with one of the error codes `SUNLS_MEM_NULL` (`S` is `NULL`) or `SUNLS_SUCCESS`.

  ```
  int SUNSPTFQMRSetMaxl(SUNLinearSolver S, int maxl);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_SPTFQMR module also includes the Fortran-callable function `FSUNSPTFQMRInit(code, pretype, maxl, ier)` to initialize this SUNLINSOL_SPTFQMR module for a given SUNDIALS solver. Here `code` is an integer input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); `pretype` and `maxl` are the same as for the C function `SUNSPTFQMR`; `ier` is an error return flag equal to 0 for success and -1 for failure. All of these input arguments should be declared so as to match C type `int`. This routine must be called *after* the NVECTOR object has been initialized. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function `FSUNMassSPTFQMRInit(pretype, maxl, ier)` initializes this SUNLINSOL_SPTFQMR module for solving mass matrix linear systems.

The `SUNSPTFQMRSetPrecType` and `SUNSPTFQMRSetMaxl` routines also support Fortran interfaces for the system and mass matrix solvers (all arguments should be commensurate with a C `int`):

- `FSUNSPTFQMRSetPrecType(code, pretype, ier)`

- `FSUNMassSPTFQMRSetPrecType(pretype, ier)`
- `FSUNSPTFQMRSetMaxl(code, maxl, ier)`
- `FSUNMassSPTFQMRSetMaxl(maxl, ier)`

8.13 The SUNLinearSolver_PCG implementation

The PCG (Preconditioned Conjugate Gradient [16]) implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_PCG, is an iterative linear solver that is designed to be compatible with any NVECTOR implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (`N_VClone`, `N_VDotProd`, `N_VScale`, `N_VLinearSum`, `N_VProd`, and `N_VDestroy`). Unlike the SPGMR and SPFGMR algorithms, PCG requires a fixed amount of memory that does not increase with the number of allowed iterations.

Unlike all of the other iterative linear solvers supplied with SUNDIALS, PCG should only be used on *symmetric* linear systems (e.g. mass matrix linear systems encountered in ARKODE). As a result, the explanation of the role of scaling and preconditioning matrices given in general must be modified in this scenario. The PCG algorithm solves a linear system $Ax = b$ where A is a symmetric ($A^T = A$), real-valued matrix. Preconditioning is allowed, and is applied in a symmetric fashion on both the right and left. Scaling is also allowed and is applied symmetrically. We denote the preconditioner and scaling matrices as follows:

- P is the preconditioner (assumed symmetric),
- S is a diagonal matrix of scale factors.

The matrices A and P are not required explicitly; only routines that provide A and P^{-1} as operators are required. The diagonal of the matrix S is held in a single NVECTOR, supplied by the user.

In this notation, PCG applies the underlying CG algorithm to the equivalent transformed system

$$\tilde{A}\tilde{x} = \tilde{b} \tag{8.3}$$

where

$$\begin{aligned} \tilde{A} &= SP^{-1}AP^{-1}S, \\ \tilde{b} &= SP^{-1}b, \\ \tilde{x} &= S^{-1}Px. \end{aligned} \tag{8.4}$$

The scaling matrix must be chosen so that the vectors $SP^{-1}b$ and $S^{-1}Px$ have dimensionless components.

The stopping test for the PCG iterations is on the L2 norm of the scaled preconditioned residual:

$$\begin{aligned} & \|\tilde{b} - \tilde{A}\tilde{x}\|_2 < \delta \\ \Leftrightarrow \\ & \|SP^{-1}b - SP^{-1}Ax\|_2 < \delta \\ \Leftrightarrow \\ & \|P^{-1}b - P^{-1}Ax\|_S < \delta \end{aligned}$$

where $\|v\|_S = \sqrt{v^T S^T S v}$, with an input tolerance δ.

The SUNLINSOL_PCG module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_PCG {
  int maxl;
  int pretype;
```

```
  int numiters;
  realtype resnorm;
  long int last_flag;
  ATimesFn ATimes;
  void* ATData;
  PSetupFn Psetup;
  PSolveFn Psolve;
  void* PData;
  N_Vector s;
  N_Vector r;
  N_Vector p;
  N_Vector z;
  N_Vector Ap;
};
```

These entries of the *content* field contain the following information:

maxl - number of PCG iterations to allow (default is 5),

pretype - flag for use of preconditioning (default is none),

numiters - number of iterations from the most-recent solve,

resnorm - final linear residual norm from the most-recent solve,

last_flag - last error return flag from an internal function,

ATimes - function pointer to perform Av product,

ATData - pointer to structure for `ATimes`,

Psetup - function pointer to preconditioner setup routine,

Psolve - function pointer to preconditioner solve routine,

PData - pointer to structure for `Psetup` and `Psolve`,

s - vector pointer for supplied scaling matrix (default is `NULL`),

r - a NVECTOR which holds the preconditioned linear system residual,

p, z, Ap - NVECTORs used for workspace by the PCG algorithm.

This solver is constructed to perform the following operations:

- During construction all NVECTOR solver data is allocated, with vectors cloned from a template NVECTOR that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLINSOL_PCG to supply the `ATimes`, `PSetup`, and `Psolve` function pointers and `s` scaling vector.
- In the "initialize" call, the solver parameters are checked for validity.
- In the "setup" call, any non-`NULL` `PSetup` function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call the PCG iteration is performed. This will include scaling and preconditioning if those options have been supplied.

The header file to include when using this module is `sunlinsol/sunlinsol_pcg.h`. The SUNLINSOL_PCG module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolpcg` module library.

The SUNLINSOL_PCG module defines implementations of all "iterative" linear solver operations listed in Table 8.2:

- `SUNLinSolGetType_PCG`
- `SUNLinSolInitialize_PCG`
- `SUNLinSolSetATimes_PCG`
- `SUNLinSolSetPreconditioner_PCG`
- `SUNLinSolSetScalingVectors_PCG` – since PCG only supports symmetric scaling, the second NVECTOR argument to this function is ignored
- `SUNLinSolSetup_PCG`
- `SUNLinSolSolve_PCG`
- `SUNLinSolNumIters_PCG`
- `SUNLinSolResNorm_PCG`
- `SUNLinSolResid_PCG`
- `SUNLinSolLastFlag_PCG`
- `SUNLinSolSpace_PCG`
- `SUNLinSolFree_PCG`

The module SUNLINSOL_PCG provides the following additional user-callable routines:

- `SUNPCG`

  This constructor function creates and allocates memory for a PCG `SUNLinearSolver`. Its arguments are an NVECTOR, a flag indicating to use preconditioning, and the number of linear iterations to allow.

  This routine will perform consistency checks to ensure that it is called with a consistent NVECTOR implementation (i.e. that it supplies the requisite vector operations). If `y` is incompatible then this routine will return `NULL`.

  A `maxl` argument that is ≤ 0 will result in the default value (5).

  Since the PCG algorithm is designed to only support symmetric preconditioning, then any of the `pretype` inputs `PREC_LEFT` (1), `PREC_RIGHT` (2), or `PREC_BOTH` (3) will result in use of the symmetric preconditioner; any other integer input will result in the default (no preconditioning). Although some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL), PCG should *only* be used with these packages when the linear systems are known to be *symmetric*. Since the scaling of matrix rows and columns must be identical in a symmetric matrix, symmetric preconditioning should work appropriately even for packages designed with one-sided preconditioning in mind.

  ```
  SUNLinearSolver SUNPCG(N_Vector y, int pretype, int maxl);
  ```

- `SUNPCGSetPrecType`

  This function updates the flag indicating use of preconditioning. As above, any one of the input values, `PREC_LEFT` (1), `PREC_RIGHT` (2), or `PREC_BOTH` (3) will enable preconditioning; `PREC_NONE` (0) disables preconditioning.

  This routine will return with one of the error codes `SUNLS_ILL_INPUT` (illegal `pretype`), `SUNLS_MEM_NULL` (`S` is `NULL`), or `SUNLS_SUCCESS`.

  ```
  int SUNPCGSetPrecType(SUNLinearSolver S, int pretype);
  ```

- SUNPCGSetMaxl

  This function updates the number of linear solver iterations to allow.

  A maxl argument that is ≤ 0 will result in the default value (5).

  This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

  ```
  int SUNPCGSetMaxl(SUNLinearSolver S, int maxl);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_PCG module also includes the Fortran-callable function FSUNPCGInit(code, pretype, maxl, ier) to initialize this SUNLINSOL_PCG module for a given SUNDIALS solver. Here code is an integer input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); pretype and maxl are the same as for the C function SUNPCG; ier is an error return flag equal to 0 for success and -1 for failure. All of these input arguments should be declared so as to match C type int. This routine must be called *after* the NVECTOR object has been initialized. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function FSUNMassPCGInit(pretype, maxl, ier) initializes this SUNLINSOL_PCG module for solving mass matrix linear systems.

The SUNPCGSetPrecType and SUNPCGSetMaxl routines also support Fortran interfaces for the system and mass matrix solvers (all arguments should be commensurate with a C int):

- FSUNPCGSetPrecType(code, pretype, ier)
- FSUNMassPCGSetPrecType(pretype, ier)
- FSUNPCGSetMaxl(code, maxl, ier)
- FSUNMassPCGSetMaxl(maxl, ier)

8.14 SUNLinearSolver Examples

There are SUNLinearSolver examples that may be installed for each implementation; these make use of the functions in test_sunlinsol.c. These example functions show simple usage of the SUNLinearSolver family of functions. The inputs to the examples depend on the linear solver type, and are output to stdout if the example is run without the appropriate number of command-line arguments.
The following is a list of the example functions in test_sunlinsol.c:

- Test_SUNLinSolGetType: Verifies the returned solver type against the value that should be returned.
- Test_SUNLinSolInitialize: Verifies that SUNLinSolInitialize can be called and returns successfully.
- Test_SUNLinSolSetup: Verifies that SUNLinSolSetup can be called and returns successfully.
- Test_SUNLinSolSolve: Given a SUNMATRIX object A, NVECTOR objects x and b (where $Ax = b$) and a desired solution tolerance tol, this routine clones x into a new vector y, calls SUNLinSolSolve to fill y as the solution to $Ay = b$ (to the input tolerance), verifies that each entry in x and y match to within 10*tol, and overwrites x with y prior to returning (in case the calling routine would like to investigate further).
- Test_SUNLinSolSetATimes (iterative solvers only): Verifies that SUNLinSolSetATimes can be called and returns successfully.
- Test_SUNLinSolSetPreconditioner (iterative solvers only): Verifies that SUNLinSolSetPreconditioner can be called and returns successfully.
- Test_SUNLinSolSetScalingVectors (iterative solvers only): Verifies that SUNLinSolSetScalingVectors can be called and returns successfully.

- `Test_SUNLinSolLastFlag`: Verifies that `SUNLinSolLastFlag` can be called, and outputs the result to `stdout`.
- `Test_SUNLinSolNumIters` (iterative solvers only): Verifies that `SUNLinSolNumIters` can be called, and outputs the result to `stdout`.
- `Test_SUNLinSolResNorm` (iterative solvers only): Verifies that `SUNLinSolResNorm` can be called, and that the result is non-negative.
- `Test_SUNLinSolResid` (iterative solvers only): Verifies that `SUNLinSolResid` can be called.
- `Test_SUNLinSolSpace` verifies that `SUNLinSolSpace` can be called, and outputs the results to `stdout`.

We'll note that these tests should be performed in a particular order. For either direct or iterative linear solvers, `Test_SUNLinSolInitialize` must be called before `Test_SUNLinSolSetup`, which must be called before `Test_SUNLinSolSolve`. Additionally, for iterative linear solvers `Test_SUNLinSolSetATimes`, `Test_SUNLinSolSetPreconditioner` and `Test_SUNLinSolSetScalingVectors` should be called before `Test_SUNLinSolInitialize`; similarly `Test_SUNLinSolNumIters`, `Test_SUNLinSolResNorm` and `Test_SUNLinSolResid` should be called after `Test_SUNLinSolSolve`. These are called in the appropriate order in all of the example problems.

8.15 SUNLinearSolver functions used by KINSOL

In Table 8.5 below, we list the linear solver functions in the SUNLINSOL module used within the KINSOL package. The table also shows, for each function, which of the code modules uses the function. In general, the main KINSOL integrator considers three categories of linear solvers, *direct*, *iterative* and *custom*, with interfaces accessible in the KINSOL header files `kinsol_direct.h` (KINDLS), `kinsol_spils.h` (KINSPILS) and `kinsol_customls.h` (KINCLS), respectively. Hence, the the table columns reference the use of SUNLINSOL functions by each of these solver interfaces.

As with the SUNMATRIX module, we emphasize that the KINSOL user does not need to know detailed usage of linear solver functions by the KINSOL code modules in order to use KINSOL. The information is presented as an implementation detail for the interested reader.

Table 8.5: List of linear solver functions usage by KINSOL code modules

	KINDLS	KINSPILS	KINCLS
SUNLinSolGetType	✓	✓	†
SUNLinSolSetATimes		✓	†
SUNLinSolSetPreconditioner		✓	†
SUNLinSolSetScalingVectors		✓	†
SUNLinSolInitialize	✓	✓	✓
SUNLinSolSetup	✓	✓	✓
SUNLinSolSolve	✓	✓	✓
SUNLinSolNumIters		✓	†
SUNLinSolResNorm		✓	†
SUNLinSolResid		✓	†
SUNLinSolLastFlag			
SUNLinSolFree			
SUNLinSolSpace	†	†	†

The linear solver functions listed in Table 8.2 with a † symbol are optionally used, in that these are only called if they are implemented in the SUNLINSOL module that is being used (i.e. their

function pointers are non-`NULL`). Also, although KINSOL does not call the `SUNLinSolLastFlag` or `SUNLinSolFree` routines directly, these are available for users to query linear solver issues and free linear solver memory directly.

Appendix A

SUNDIALS Package Installation Procedure

The installation of any SUNDIALS package is accomplished by installing the SUNDIALS suite as a whole, according to the instructions that follow. The same procedure applies whether or not the downloaded file contains one or all solvers in SUNDIALS.

The SUNDIALS suite (or individual solvers) are distributed as compressed archives (`.tar.gz`). The name of the distribution archive is of the form *solver*`-x.y.z.tar.gz`, where *solver* is one of: `sundials`, `cvode`, `cvodes`, `arkode`, `ida`, `idas`, or `kinsol`, and `x.y.z` represents the version number (of the SUNDIALS suite or of the individual solver) . To begin the installation, first uncompress and expand the sources, by issuing

```
% tar xzf solver-x.y.z.tar.gz
```

This will extract source files under a directory *solver*`-x.y.z`.

Starting with version `2.6.0` of SUNDIALS, CMake is the only supported method of installation. The explanations of the installation procedure begins with a few common observations:

- The remainder of this chapter will follow these conventions:

  srcdir is the directory *solver*`-x.y.z` created above; i.e., the directory containing the SUNDIALS sources.

  builddir is the (temporary) directory under which SUNDIALS is built.

  instdir is the directory under which the SUNDIALS exported header files and libraries will be installed. Typically, header files are exported under a directory *instdir*`/include` while libraries are installed under *instdir*`/lib`, with *instdir* specified at configuration time.

- For SUNDIALS CMake-based installation, in-source builds are prohibited; in other words, the build directory *builddir* can **not** be the same as *srcdir* and such an attempt will lead to an error. This prevents "polluting" the source tree and allows efficient builds for different configurations and/or options.

- The installation directory *instdir* can **not** be the same as the source directory *srcdir*.

- By default, only the libraries and header files are exported to the installation directory *instdir*. If enabled by the user (with the appropriate toggle for CMake), the examples distributed with SUNDIALS will be built together with the solver libraries but the installation step will result in exporting (by default in a subdirectory of the installation directory) the example sources and sample outputs together with automatically generated configuration files that reference the *installed* SUNDIALS headers and libraries. As such, these configuration files for the SUNDIALS examples can be used as "templates" for your own problems. CMake installs `CMakeLists.txt` files and also (as an option available only under Unix/Linux) `Makefile` files. Note this installation

approach also allows the option of building the SUNDIALS examples without having to install them. (This can be used as a sanity check for the freshly built libraries.)

- Even if generation of shared libraries is enabled, only static libraries are created for the FCMIX modules. (Because of the use of fixed names for the Fortran user-provided subroutines, FCMIX shared libraries would result in "undefined symbol" errors at link time.)

A.1 CMake-based installation

CMake-based installation provides a platform-independent build system. CMake can generate Unix and Linux Makefiles, as well as KDevelop, Visual Studio, and (Apple) XCode project files from the same configuration file. In addition, CMake also provides a GUI front end and which allows an interactive build and installation process.

The SUNDIALS build process requires CMake version `2.8.1` or higher and a working C compiler. On Unix-like operating systems, it also requires Make (and `curses`, including its development libraries, for the GUI front end to CMake, `ccmake`), while on Windows it requires Visual Studio. While many Linux distributions offer CMake, the version included is probably out of date. Many new CMake features have been added recently, and you should download the latest version from `http://www.cmake.org`. Build instructions for CMake (only necessary for Unix-like systems) can be found on the CMake website. Once CMake is installed, Linux/Unix users will be able to use `ccmake`, while Windows users will be able to use `CMakeSetup`.

As previously noted, when using CMake to configure, build and install SUNDIALS, it is always required to use a separate build directory. While in-source builds are possible, they are explicitly prohibited by the SUNDIALS CMake scripts (one of the reasons being that, unlike autotools, CMake does not provide a `make distclean` procedure and it is therefore difficult to clean-up the source tree after an in-source build). By ensuring a separate build directory, it is an easy task for the user to clean-up all traces of the build by simply removing the build directory. CMake does generate a `make clean` which will remove files generated by the compiler and linker.

A.1.1 Configuring, building, and installing on Unix-like systems

The default CMake configuration will build all included solvers and associated examples and will build static and shared libraries. The *installdir* defaults to `/usr/local` and can be changed by setting the `CMAKE_INSTALL_PREFIX` variable. Support for FORTRAN and all other options are disabled.

CMake can be used from the command line with the `cmake` command, or from a `curses`-based GUI by using the `ccmake` command. Examples for using both methods will be presented. For the examples shown it is assumed that there is a top level SUNDIALS directory with appropriate source, build and install directories:

```
% mkdir (...)sundials/instdir
% mkdir (...)sundials/builddir
% cd (...)sundials/builddir
```

Building with the GUI

Using CMake with the GUI follows this general process:

- Select and modify values, run configure (`c` key)
- New values are denoted with an asterisk
- To set a variable, move the cursor to the variable and press enter
  - If it is a boolean (ON/OFF) it will toggle the value
  - If it is string or file, it will allow editing of the string

  - For file and directories, the `<tab>` key can be used to complete

- Repeat until all values are set as desired and the generate option is available (`g` key)
- Some variables (advanced variables) are not visible right away
- To see advanced variables, toggle to advanced mode (`t` key)
- To search for a variable press `/` key, and to repeat the search, press the `n` key

To build the default configuration using the GUI, from the *builddir* enter the ccmake command and point to the *srcdir*:

```
% ccmake ../srcdir
```

The default configuration screen is shown in Figure A.1.

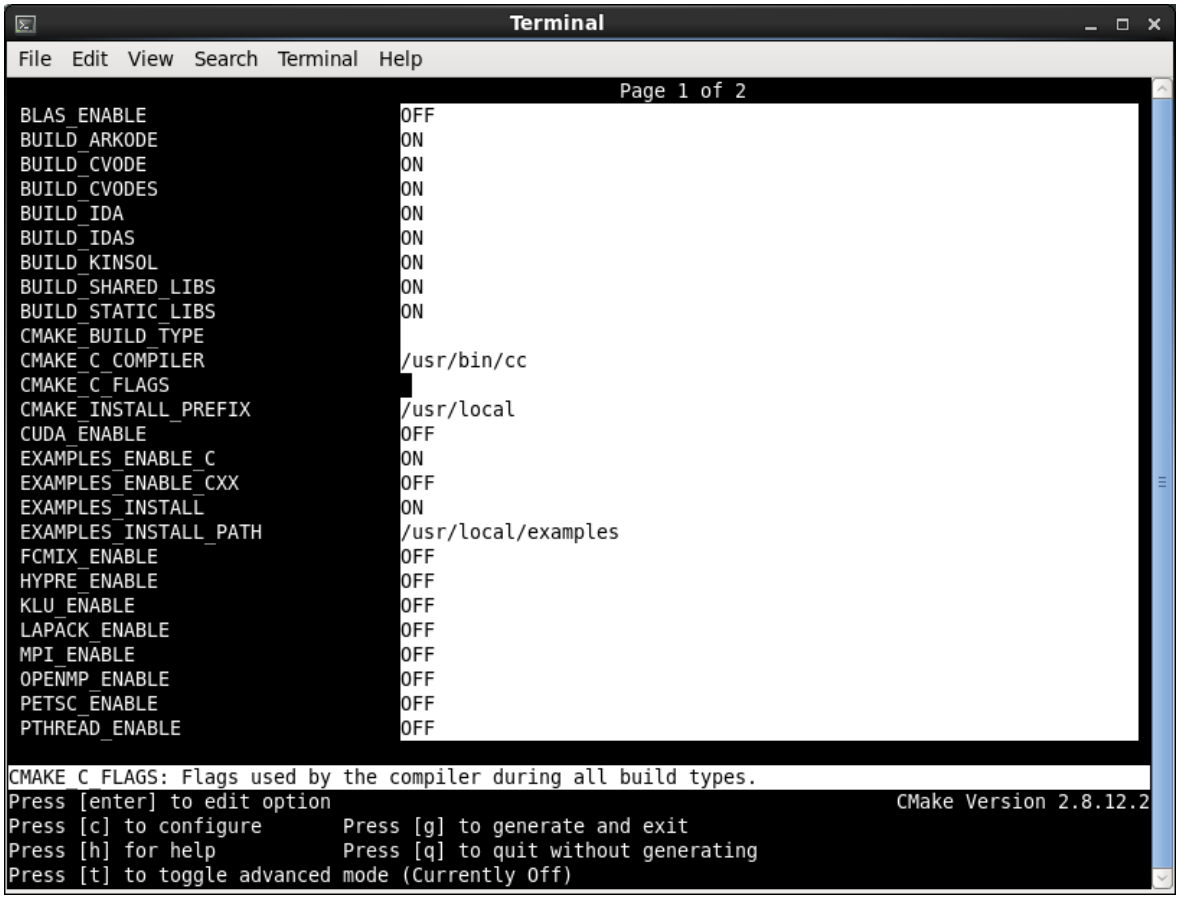


Figure A.1: Default configuration screen. Note: Initial screen is empty. To get this default configuration, press 'c' repeatedly (accepting default values denoted with asterisk) until the 'g' option is available.

The default *instdir* for both SUNDIALS and corresponding examples can be changed by setting the `CMAKE_INSTALL_PREFIX` and the `EXAMPLES_INSTALL_PATH` as shown in figure A.2.

Pressing the (`g` key) will generate makefiles including all dependencies and all rules to build SUNDIALS on this system. Back at the command prompt, you can now run:

```
% make
```

To install SUNDIALS in the installation directory specified in the configuration, simply run:

```
% make install
```

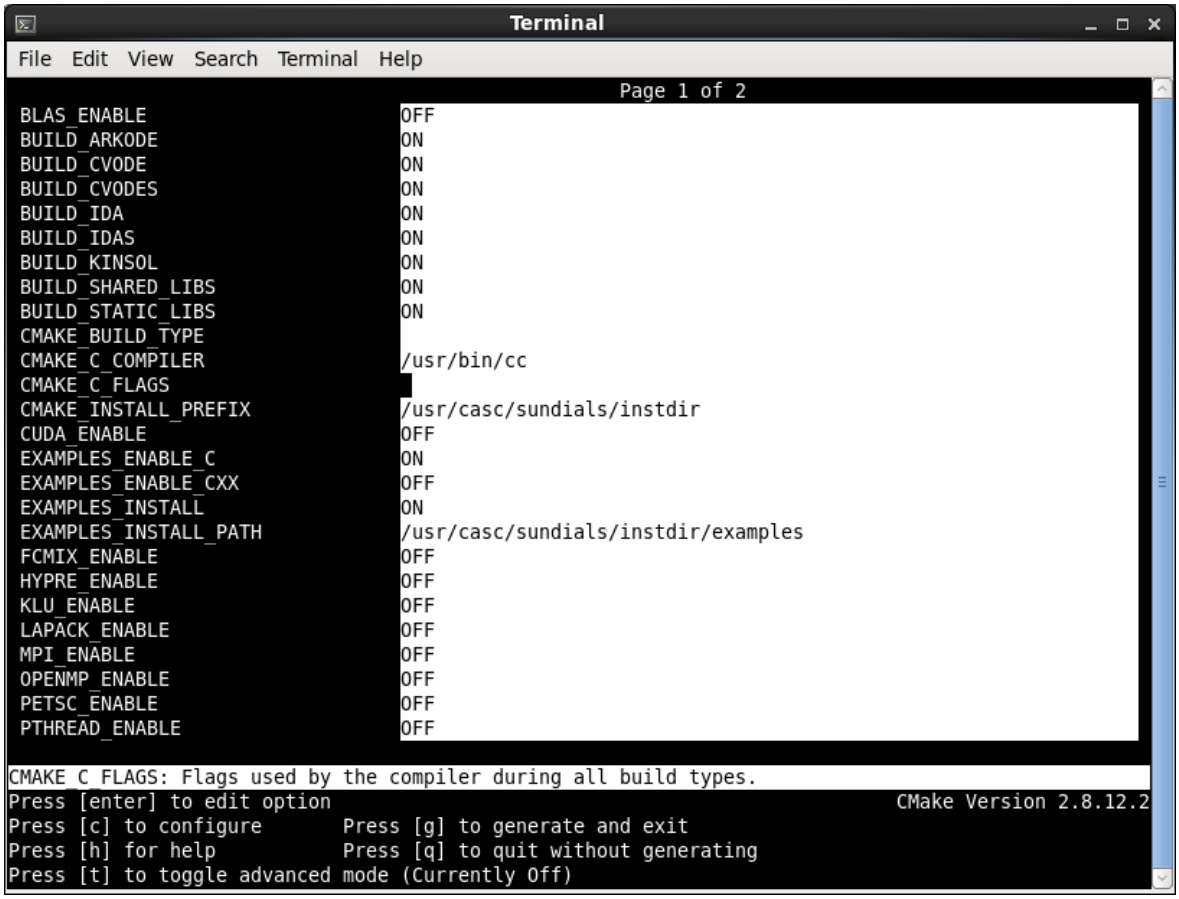


Figure A.2: Changing the *instdir* for SUNDIALS and corresponding examples

Building from the command line

Using CMake from the command line is simply a matter of specifying CMake variable settings with the cmake command. The following will build the default configuration:

```
% cmake -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> ../srcdir
% make
% make install
```

A.1.2 Configuration options (Unix/Linux)

A complete list of all available options for a CMake-based SUNDIALS configuration is provide below. Note that the default values shown are for a typical configuration on a Linux system and are provided as illustration only.

`BLAS_ENABLE` - Enable BLAS support
Default: OFF
Note: Setting this option to ON will trigger additional CMake options. See additional information on building with BLAS enabled in A.1.4.

`BLAS_LIBRARIES` - BLAS library
Default: /usr/lib/libblas.so

Note: CMake will search for libraries in your `LD_LIBRARY_PATH` prior to searching default system paths.

`BUILD_ARKODE` - Build the ARKODE library
Default: ON

`BUILD_CVODE` - Build the CVODE library
Default: ON

`BUILD_CVODES` - Build the CVODES library
Default: ON

`BUILD_IDA` - Build the IDA library
Default: ON

`BUILD_IDAS` - Build the IDAS library
Default: ON

`BUILD_KINSOL` - Build the KINSOL library
Default: ON

`BUILD_SHARED_LIBS` - Build shared libraries
Default: ON

`BUILD_STATIC_LIBS` - Build static libraries
Default: ON

`CMAKE_BUILD_TYPE` - Choose the type of build, options are: `None` (CMAKE_C_FLAGS used), `Debug`, `Release`, `RelWithDebInfo`, and `MinSizeRel`
Default:
Note: Specifying a build type will trigger the corresponding build type specific compiler flag options below which will be appended to the flags set by CMAKE_<language>_FLAGS.

`CMAKE_C_COMPILER` - C compiler
Default: /usr/bin/cc

`CMAKE_C_FLAGS` - Flags for C compiler
Default:

`CMAKE_C_FLAGS_DEBUG` - Flags used by the C compiler during debug builds
Default: -g

`CMAKE_C_FLAGS_MINSIZEREL` - Flags used by the C compiler during release minsize builds
Default: -Os -DNDEBUG

`CMAKE_C_FLAGS_RELEASE` - Flags used by the C compiler during release builds
Default: -O3 -DNDEBUG

`CMAKE_CXX_COMPILER` - C++ compiler
Default: /usr/bin/c++
Note: A C++ compiler (and all related options) are only triggered if C++ examples are enabled (`EXAMPLES_ENABLE_CXX` is ON). All SUNDIALS solvers can be used from C++ applications by default without setting any additional configuration options.

`CMAKE_CXX_FLAGS` - Flags for C++ compiler
Default:

`CMAKE_CXX_FLAGS_DEBUG` - Flags used by the C++ compiler during debug builds
Default: -g

CMAKE_CXX_FLAGS_MINSIZEREL - Flags used by the C++ compiler during release minsize builds
Default: -Os -DNDEBUG

CMAKE_CXX_FLAGS_RELEASE - Flags used by the C++ compiler during release builds
Default: -O3 -DNDEBUG

CMAKE_Fortran_COMPILER - Fortran compiler
Default: /usr/bin/gfortran
Note: Fortran support (and all related options) are triggered only if either Fortran-C support is enabled (FCMIX_ENABLE is ON) or BLAS/LAPACK support is enabled (BLAS_ENABLE or LAPACK_ENABLE is ON).

CMAKE_Fortran_FLAGS - Flags for Fortran compiler
Default:

CMAKE_Fortran_FLAGS_DEBUG - Flags used by the Fortran compiler during debug builds
Default: -g

CMAKE_Fortran_FLAGS_MINSIZEREL - Flags used by the Fortran compiler during release minsize builds
Default: -Os

CMAKE_Fortran_FLAGS_RELEASE - Flags used by the Fortran compiler during release builds
Default: -O3

CMAKE_INSTALL_PREFIX - Install path prefix, prepended onto install directories
Default: /usr/local
Note: The user must have write access to the location specified through this option. Exported SUNDIALS header files and libraries will be installed under subdirectories include and lib of CMAKE_INSTALL_PREFIX, respectively.

CUDA_ENABLE - Build the SUNDIALS CUDA vector module.
Default: OFF

EXAMPLES_ENABLE_C - Build the SUNDIALS C examples
Default: ON

EXAMPLES_ENABLE_CUDA - Build the SUNDIALS CUDA examples
Default: OFF
Note: You need to enable CUDA support to build these examples.

EXAMPLES_ENABLE_CXX - Build the SUNDIALS C++ examples
Default: OFF

EXAMPLES_ENABLE_RAJA - Build the SUNDIALS RAJA examples
Default: OFF
Note: You need to enable CUDA and RAJA support to build these examples.

EXAMPLES_ENABLE_F77 - Build the SUNDIALS Fortran77 examples
Default: ON (if FCMIX_ENABLE is ON)

EXAMPLES_ENABLE_F90 - Build the SUNDIALS Fortran90 examples
Default: OFF

EXAMPLES_INSTALL - Install example files
Default: ON
Note: This option is triggered when any of the SUNDIALS example programs are enabled (EXAMPLES_ENABLE_<language> is ON). If the user requires installation of example programs then the sources and sample output files for all SUNDIALS modules that are currently enabled will be exported to the directory specified by EXAMPLES_INSTALL_PATH. A CMake configuration

script will also be automatically generated and exported to the same directory. Additionally, if the configuration is done under a Unix-like system, makefiles for the compilation of the example programs (using the installed SUNDIALS libraries) will be automatically generated and exported to the directory specified by `EXAMPLES_INSTALL_PATH`.

`EXAMPLES_INSTALL_PATH` - Output directory for installing example files
Default: /usr/local/examples
Note: The actual default value for this option will be an `examples` subdirectory created under `CMAKE_INSTALL_PREFIX`.

`FCMIX_ENABLE` - Enable Fortran-C support
Default: OFF

`HYPRE_ENABLE` - Enable *hypre* support
Default: OFF
Note: See additional information on building with *hypre* enabled in A.1.4.

`HYPRE_INCLUDE_DIR` - Path to *hypre* header files

`HYPRE_LIBRARY_DIR` - Path to *hypre* installed library files

`KLU_ENABLE` - Enable KLU support
Default: OFF
Note: See additional information on building with KLU enabled in A.1.4.

`KLU_INCLUDE_DIR` - Path to SuiteSparse header files

`KLU_LIBRARY_DIR` - Path to SuiteSparse installed library files

`LAPACK_ENABLE` - Enable LAPACK support
Default: OFF
Note: Setting this option to ON will trigger additional CMake options. See additional information on building with LAPACK enabled in A.1.4.

`LAPACK_LIBRARIES` - LAPACK (and BLAS) libraries
Default: /usr/lib/liblapack.so;/usr/lib/libblas.so
Note: CMake will search for libraries in your `LD_LIBRARY_PATH` prior to searching default system paths.

`MPI_ENABLE` - Enable MPI support (build the parallel nvector).
Default: OFF
Note: Setting this option to ON will trigger several additional options related to MPI.

`MPI_MPICC` - `mpicc` program
Default:

`MPI_MPICXX` - `mpicxx` program
Default:
Note: This option is triggered only if MPI is enabled (`MPI_ENABLE` is ON) and C++ examples are enabled (`EXAMPLES_ENABLE_CXX` is ON). All SUNDIALS solvers can be used from C++ MPI applications by default without setting any additional configuration options other than `MPI_ENABLE`.

`MPI_MPIF77` - `mpif77` program
Default:
Note: This option is triggered only if MPI is enabled (`MPI_ENABLE` is ON) and Fortran-C support is enabled (`FCMIX_ENABLE` is ON).

MPI_MPIF90 - mpif90 program
Default:
Note: This option is triggered only if MPI is enabled (MPI_ENABLE is ON), Fortran-C support is enabled (FCMIX_ENABLE is ON), and Fortran90 examples are enabled (EXAMPLES_ENABLE_F90 is ON).

MPI_RUN_COMMAND - Specify run command for MPI
Default: mpirun Note: This option is triggered only if MPI is enabled (MPI_ENABLE is ON).

OPENMP_ENABLE - Enable OpenMP support (build the OpenMP nvector).
Default: OFF

PETSC_ENABLE - Enable PETSc support
Default: OFF
Note: See additional information on building with PETSc enabled in A.1.4.

PETSC_INCLUDE_DIR - Path to PETSc header files

PETSC_LIBRARY_DIR - Path to PETSc installed library files

PTHREAD_ENABLE - Enable Pthreads support (build the Pthreads nvector).
Default: OFF

RAJA_ENABLE - Enable RAJA support (build the RAJA nvector).
Default: OFF
Note: You need to enable CUDA in order to build the RAJA vector module.

SUNDIALS_INDEX_TYPE - Integer type used for SUNDIALS indices, options are: int32_t or int64_t
Default: int64_t

SUNDIALS_PRECISION - Precision used in SUNDIALS, options are: double, single, or extended
Default: double

SUPERLUMT_ENABLE - Enable SuperLU_MT support
Default: OFF
Note: See additional information on building with SuperLU_MT enabled in A.1.4.

SUPERLUMT_INCLUDE_DIR - Path to SuperLU_MT header files (typically SRC directory)

SUPERLUMT_LIBRARY_DIR - Path to SuperLU_MT installed library files

SUPERLUMT_THREAD_TYPE - Must be set to Pthread or OpenMP
Default: Pthread

USE_GENERIC_MATH - Use generic (stdc) math libraries
Default: ON

xSDK Configuration Options

SUNDIALS supports CMake configuration options defined by the Extreme-scale Scientific Software Development Kit (xSDK) community policies (see https://xsdk.info for more information). xSDK CMake options are unused by default but may be activated by setting USE_XSDK_DEFAULTS to ON.

When xSDK options are active, they will overwrite the corresponding SUNDIALS option and may have different default values (see details below). As such the equivalent SUNDIALS options should not be used when configuring with xSDK options. In the GUI front end to CMake (ccmake), setting USE_XSDK_DEFAULTS to ON will hide the corresponding SUNDIALS options as advanced CMake variables. During configuration, messages are output detailing which xSDK flags are active and the equivalent SUNDIALS options that are replaced. Below is a complete list xSDK options and the corresponding SUNDIALS options if applicable.

TPL_BLAS_LIBRARIES - BLAS library
Default: /usr/lib/libblas.so
SUNDIALS equivalent: BLAS_LIBRARIES
Note: CMake will search for libraries in your LD_LIBRARY_PATH prior to searching default system paths.

TPL_ENABLE_BLAS - Enable BLAS support
Default: OFF
SUNDIALS equivalent: BLAS_ENABLE

TPL_ENABLE_HYPRE - Enable *hypre* support
Default: OFF
SUNDIALS equivalent: HYPRE_ENABLE

TPL_ENABLE_KLU - Enable KLU support
Default: OFF
SUNDIALS equivalent: KLU_ENABLE

TPL_ENABLE_PETSC - Enable PETSc support
Default: OFF
SUNDIALS equivalent: PETSC_ENABLE

TPL_ENABLE_LAPACK - Enable LAPACK support
Default: OFF
SUNDIALS equivalent: LAPACK_ENABLE

TPL_ENABLE_SUPERLUMT - Enable SuperLU_MT support
Default: OFF
SUNDIALS equivalent: SUPERLUMT_ENABLE

TPL_HYPRE_INCLUDE_DIRS - Path to *hypre* header files
SUNDIALS equivalent: HYPRE_INCLUDE_DIR

TPL_HYPRE_LIBRARIES - *hypre* library
SUNDIALS equivalent: N/A

TPL_KLU_INCLUDE_DIRS - Path to KLU header files
SUNDIALS equivalent: KLU_INCLUDE_DIR

TPL_KLU_LIBRARIES - KLU library
SUNDIALS equivalent: N/A

TPL_LAPACK_LIBRARIES - LAPACK (and BLAS) libraries
Default: /usr/lib/liblapack.so;/usr/lib/libblas.so
SUNDIALS equivalent: LAPACK_LIBRARIES
Note: CMake will search for libraries in your LD_LIBRARY_PATH prior to searching default system paths.

TPL_PETSC_INCLUDE_DIRS - Path to PETSc header files
SUNDIALS equivalent: PETSC_INCLUDE_DIR

TPL_PETSC_LIBRARIES - PETSc library
SUNDIALS equivalent: N/A

TPL_SUPERLUMT_INCLUDE_DIRS - Path to SuperLU_MT header files
SUNDIALS equivalent: SUPERLUMT_INCLUDE_DIR

TPL_SUPERLUMT_LIBRARIES - SuperLU_MT library
SUNDIALS equivalent: N/A

TPL_SUPERLUMT_THREAD_TYPE - SuperLU_MT library thread type
SUNDIALS equivalent: SUPERLUMT_THREAD_TYPE

USE_XSDK_DEFAULTS - Enable xSDK default configuration settings
Default: OFF
SUNDIALS equivalent: N/A
Note: Enabling xSDK defaults also sets CMAKE_BUILD_TYPE to Debug

XSDK_ENABLE_FORTRAN - Enable SUNDIALS Fortran interface
Default: OFF
SUNDIALS equivalent: FCMIX_ENABLE

XSDK_INDEX_SIZE - Integer size (bits) used for indices in SUNDIALS, options are: 32 or 64
Default: 32
SUNDIALS equivalent: SUNDIALS_INDEX_TYPE

XSDK_PRECISION - Precision used in SUNDIALS, options are: double, single, or quad
Default: double
SUNDIALS equivalent: SUNDIALS_PRECISION

A.1.3 Configuration examples

The following examples will help demonstrate usage of the CMake configure options.
To configure SUNDIALS using the default C and Fortran compilers, and default mpicc and mpif77 parallel compilers, enable compilation of examples, and install libraries, headers, and example sources under subdirectories of /home/myname/sundials/, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DMPI_ENABLE=ON \
> -DFCMIX_ENABLE=ON \
> /home/myname/sundials/srcdir
%
% make install
%
```

To disable installation of the examples, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DMPI_ENABLE=ON \
> -DFCMIX_ENABLE=ON \
> -DEXAMPLES_INSTALL=OFF \
> /home/myname/sundials/srcdir
%
% make install
%
```

A.1.4 Working with external Libraries

The SUNDIALS suite contains many options to enable implementation flexibility when developing solutions. The following are some notes addressing specific configurations when using the supported third party libraries. When building SUNDIALS as a shared library external libraries any used with SUNDIALS must also be build as a shared library or as a static library compiled with the -fPIC flag.

Building with BLAS

SUNDIALS does not utilize BLAS directly but it may be needed by other external libraries that SUNDIALS can be build with (e.g. LAPACK, PETSc, SuperLU_MT, etc.). To enable BLAS, set the BLAS_ENABLE option to ON. If the directory containing the BLAS library is in the LD_LIBRARY_PATH environment variable, CMake will set the BLAS_LIBRARIES variable accordingly, otherwise CMake will attempt to find the BLAS library in standard system locations. To explicitly tell CMake what libraries to use, the BLAS_LIBRARIES variable can be set to the desired library. Example:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DBLAS_ENABLE=ON \
> -DBLAS_LIBRARIES=/myblaspath/lib/libblas.so \
> -DSUPERLUMT_ENABLE=ON \
> -DSUPERLUMT_INCLUDE_DIR=/mysuperlumtpath/SRC
> -DSUPERLUMT_LIBRARY_DIR=/mysuperlumtpath/lib
> /home/myname/sundials/srcdir
%
% make install
%
```

If enabling LAPACK and allowing CMake to automatically locate the LAPACK library, it is not necessary to also enable BLAS as CMake will find the corresponding BLAS library and include it when searching for LAPACK.

Building with LAPACK

To enable LAPACK, set the LAPACK_ENABLE option to ON. If the directory containing the LAPACK library is in the LD_LIBRARY_PATH environment variable, CMake will set the LAPACK_LIBRARIES variable accordingly, otherwise CMake will attempt to find the LAPACK library in standard system locations. To explicitly tell CMake what library to use, the LAPACK_LIBRARIES variable can be set to the desired libraries. When setting the LAPACK location explicitly the location of the corresponding BLAS library will also need to be set. Example:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DBLAS_ENABLE=ON \
> -DBLAS_LIBRARIES=/mylapackpath/lib/libblas.so \
> -DLAPACK_ENABLE=ON \
> -DLAPACK_LIBRARIES=/mylapackpath/lib/liblapack.so \
> /home/myname/sundials/srcdir
%
% make install
%
```

If enabling LAPACK and allowing CMake to automatically locate the LAPACK library, it is not necessary to also enable BLAS as CMake will find the corresponding BLAS library and include it when searching for LAPACK.

Building with KLU

The KLU libraries are part of SuiteSparse, a suite of sparse matrix software, available from the Texas A&M University website: http://faculty.cse.tamu.edu/davis/suitesparse.html. SUNDIALS has been tested with SuiteSparse version 4.5.3. To enable KLU, set KLU_ENABLE to ON, set KLU_INCLUDE_DIR to the include path of the KLU installation and set KLU_LIBRARY_DIR to the lib path of the KLU

installation. The CMake configure will result in populating the following variables: AMD_LIBRARY, AMD_LIBRARY_DIR, BTF_LIBRARY, BTF_LIBRARY_DIR, COLAMD_LIBRARY, COLAMD_LIBRARY_DIR, and KLU_LIBRARY.

Building with SuperLU_MT

The SuperLU_MT libraries are available for download from the Lawrence Berkeley National Laboratory website: http://crd-legacy.lbl.gov/~xiaoye/SuperLU/#superlu_mt. SUNDIALS has been tested with SuperLU_MT version 3.1. To enable SuperLU_MT, set SUPERLUMT_ENABLE to ON, set SUPERLUMT_INCLUDE_DIR to the SRC path of the SuperLU_MT installation, and set the variable SUPERLUMT_LIBRARY_DIR to the lib path of the SuperLU_MT installation. At the same time, the variable SUPERLUMT_THREAD_TYPE must be set to either Pthread or OpenMP.

Do not mix thread types when building SUNDIALS solvers. If threading is enabled for SUNDIALS by having either OPENMP_ENABLE or PTHREAD_ENABLE set to ON then SuperLU_MT should be set to use the same threading type.

Building with PETSc

The PETSc libraries are available for download from the Argonne National Laboratory website: http://www.mcs.anl.gov/petsc. SUNDIALS has been tested with PETSc version 3.7.2. To enable PETSc, set PETSC_ENABLE to ON, set PETSC_INCLUDE_DIR to the include path of the PETSc installation, and set the variable PETSC_LIBRARY_DIR to the lib path of the PETSc installation.

Building with *hypre*

The *hypre* libraries are available for download from the Lawrence Livermore National Laboratory website: http://computation.llnl.gov/projects/hypre. SUNDIALS has been tested with *hypre* version 2.11.1. To enable *hypre*, set HYPRE_ENABLE to ON, set HYPRE_INCLUDE_DIR to the include path of the *hypre* installation, and set the variable HYPRE_LIBRARY_DIR to the lib path of the *hypre* installation.

Building with CUDA

SUNDIALS CUDA modules and examples are tested with version 8.0 of the CUDA toolkit. To build them, you need to install the Toolkit and compatible NVIDIA drivers. Both are available for download from NVIDIA website: https://developer.nvidia.com/cuda-downloads. To enable CUDA, set CUDA_ENABLE to ON. If you installed CUDA in a nonstandard location, you may be prompted to set the variable CUDA_TOOLKIT_ROOT_DIR with your CUDA Toolkit installation path. To enable CUDA examples, set EXAMPLES_ENABLE_CUDA to ON.

Building with RAJA

To build SUNDIALS RAJA modules you need to enable SUNDIALS CUDA support, first. You also need a CUDA-enabled RAJA installation on your system. RAJA is free software, developed by Lawrence Livermore National Laboratory, and can be obtained from https://github.com/LLNL/RAJA. Next you need to set RAJA_ENABLE to ON, to enable building the RAJA vector module, and EXAMPLES_ENABLE_RAJA to ON to build the RAJA examples. If you installed RAJA to a nonstandard location you will be prompted to set the variable RAJA_DIR with the path to the RAJA CMake configuration file. SUNDIALS was tested with RAJA version 0.3.

A.2 Building and Running Examples

Each of the SUNDIALS solvers is distributed with a set of examples demonstrating basic usage. To build and install the examples, set at least of the EXAMPLES_ENABLE_<language> options to ON, and set EXAMPLES_INSTALL to ON. Specify the installation path for the examples with the variable

EXAMPLES_INSTALL_PATH. CMake will generate CMakeLists.txt configuration files (and Makefile files if on Linux/Unix) that reference the *installed* SUNDIALS headers and libraries.

Either the CMakeLists.txt file or the traditional Makefile may be used to build the examples as well as serve as a template for creating user developed solutions. To use the supplied Makefile simply run make to compile and generate the executables. To use CMake from within the installed example directory, run cmake (or ccmake to use the GUI) followed by make to compile the example code. Note that if CMake is used, it will overwrite the traditional Makefile with a new CMake-generated Makefile. The resulting output from running the examples can be compared with example output bundled in the SUNDIALS distribution.

NOTE: There will potentially be differences in the output due to machine architecture, compiler versions, use of third party libraries etc.

A.3 Configuring, building, and installing on Windows

CMake can also be used to build SUNDIALS on Windows. To build SUNDIALS for use with Visual Studio the following steps should be performed:

1. Unzip the downloaded tar file(s) into a directory. This will be the *srcdir*
2. Create a separate *builddir*
3. Open a Visual Studio Command Prompt and cd to *builddir*
4. Run cmake-gui ../*srcdir*
   (a) Hit Configure
   (b) Check/Uncheck solvers to be built
   (c) Change CMAKE_INSTALL_PREFIX to *instdir*
   (d) Set other options as desired
   (e) Hit Generate
5. Back in the VS Command Window:
   (a) Run msbuild ALL_BUILD.vcxproj
   (b) Run msbuild INSTALL.vcxproj

The resulting libraries will be in the *instdir*. The SUNDIALS project can also now be opened in Visual Studio. Double click on the ALL_BUILD.vcxproj file to open the project. Build the whole *solution* to create the SUNDIALS libraries. To use the SUNDIALS libraries in your own projects, you must set the include directories for your project, add the SUNDIALS libraries to your project solution, and set the SUNDIALS libraries as dependencies for your project.

A.4 Installed libraries and exported header files

Using the CMake SUNDIALS build system, the command

```
% make install
```

will install the libraries under *libdir* and the public header files under *includedir*. The values for these directories are *instdir*/lib and *instdir*/include, respectively. The location can be changed by setting the CMake variable CMAKE_INSTALL_PREFIX. Although all installed libraries reside under *libdir*/lib, the public header files are further organized into subdirectories under *includedir*/include.

The installed libraries and exported header files are listed for reference in Table A.1. The file extension *.lib* is typically .so for shared libraries and .a for static libraries. Note that, in the Tables, names are relative to *libdir* for libraries and to *includedir* for header files.

A typical user program need not explicitly include any of the shared SUNDIALS header files from under the *includedir*`/include/sundials` directory since they are explicitly included by the appropriate solver header files (*e.g.*, `cvode_dense.h` includes `sundials_dense.h`). However, it is both legal and safe to do so, and would be useful, for example, if the functions declared in `sundials_dense.h` are to be used in building a preconditioner.

Table A.1: SUNDIALS libraries and header files

SHARED	Libraries	n/a	
	Header files	sundials/sundials_config.h	sundials/sundials_fconfig.h
		sundials/sundials_types.h	sundials/sundials_math.h
		sundials/sundials_nvector.h	sundials/sundials_fnvector.h
		sundials/sundials_iterative.h	sundials/sundials_direct.h
		sundials/sundials_dense.h	sundials/sundials_band.h
		sundials/sundials_matrix.h	sundials/sundials_version.h
		sundials/sundials_linearsolver.h	
NVECTOR_SERIAL	Libraries	libsundials_nvecserial.*lib*	libsundials_fnvecserial.a
	Header files	nvector/nvector_serial.h	
NVECTOR_PARALLEL	Libraries	libsundials_nvecparallel.*lib*	libsundials_fnvecparallel.a
	Header files	nvector/nvector_parallel.h	
NVECTOR_OPENMP	Libraries	libsundials_nvecopenmp.*lib*	libsundials_fnvecopenmp.a
	Header files	nvector/nvector_openmp.h	
NVECTOR_PTHREADS	Libraries	libsundials_nvecpthreads.*lib*	libsundials_fnvecpthreads.a
	Header files	nvector/nvector_pthreads.h	
NVECTOR_PARHYP	Libraries	libsundials_nvecparhyp.*lib*	
	Header files	nvector/nvector_parhyp.h	
NVECTOR_PETSC	Libraries	libsundials_nvecpetsc.*lib*	
	Header files	nvector/nvector_petsc.h	
NVECTOR_CUDA	Libraries	libsundials_nveccuda.*lib*	
	Header files	nvector/nvector_cuda.h	
		nvector/cuda/ThreadPartitioning.hpp	
		nvector/cuda/Vector.hpp	
		nvector/cuda/VectorKernels.cuh	
NVECTOR_RAJA	Libraries	libsundials_nvecraja.*lib*	
	Header files	nvector/nvector_raja.h	
		nvector/raja/Vector.hpp	
SUNMATRIX_BAND	Libraries	libsundials_sunmatrixband.*lib*	
		libsundials_fsunmatrixband.a	
	Header files	sunmatrix/sunmatrix_band.h	
SUNMATRIX_DENSE	Libraries	libsundials_sunmatrixdense.*lib*	
		libsundials_fsunmatrixdense.a	
	Header files	sunmatrix/sunmatrix_dense.h	
SUNMATRIX_SPARSE	Libraries	libsundials_sunmatrixsparse.*lib*	

continued on next page

continued from last page			
		libsundials_fsunmatrixsparse.a	
	Header files	sunmatrix/sunmatrix_sparse.h	
SUNLINSOL_BAND	Libraries	libsundials_sunlinsolband.*lib* libsundials_fsunlinsolband.a	
	Header files	sunlinsol/sunlinsol_band.h	
SUNLINSOL_DENSE	Libraries	libsundials_sunlinsoldense.*lib* libsundials_fsunlinsoldense.a	
	Header files	sunlinsol/sunlinsol_dense.h	
SUNLINSOL_KLU	Libraries	libsundials_sunlinsolklu.*lib* libsundials_fsunlinsolklu.a	
	Header files	sunlinsol/sunlinsol_klu.h	
SUNLINSOL_LAPACKBAND	Libraries	libsundials_sunlinsollapackband.*lib* libsundials_fsunlinsollapackband.a	
	Header files	sunlinsol/sunlinsol_lapackband.h	
SUNLINSOL_LAPACKDENSE	Libraries	libsundials_sunlinsollapackdense.*lib* libsundials_fsunlinsollapackdense.a	
	Header files	sunlinsol/sunlinsol_lapackdense.h	
SUNLINSOL_PCG	Libraries	libsundials_sunlinsolpcg.*lib* libsundials_fsunlinsolpcg.a	
	Header files	sunlinsol/sunlinsol_pcg.h	
SUNLINSOL_SPBCGS	Libraries	libsundials_sunlinsolspbcgs.*lib* libsundials_fsunlinsolspbcgs.a	
	Header files	sunlinsol/sunlinsol_spbcgs.h	
SUNLINSOL_SPFGMR	Libraries	libsundials_sunlinsolspfgmr.*lib* libsundials_fsunlinsolspfgmr.a	
	Header files	sunlinsol/sunlinsol_spfgmr.h	
SUNLINSOL_SPGMR	Libraries	libsundials_sunlinsolspgmr.*lib* libsundials_fsunlinsolspgmr.a	
	Header files	sunlinsol/sunlinsol_spgmr.h	
SUNLINSOL_SPTFQMR	Libraries	libsundials_sunlinsolsptfqmr.*lib* libsundials_fsunlinsolsptfqmr.a	
	Header files	sunlinsol/sunlinsol_sptfqmr.h	
SUNLINSOL_SUPERLUMT	Libraries	libsundials_sunlinsolsuperlumt.*lib* libsundials_fsunlinsolsuperlumt.a	
	Header files	sunlinsol/sunlinsol_superlumt.h	
CVODE	Libraries	libsundials_cvode.*lib*	libsundials_fcvode.a
	Header files	cvode/cvode.h cvode/cvode_direct.h cvode/cvode_bandpre.h	cvode/cvode_impl.h cvode/cvode_spils.h cvode/cvode_bbdpre.h
CVODES	Libraries	libsundials_cvodes.*lib*	
	Header files	cvodes/cvodes.h	cvodes/cvodes_impl.h
continued on next page			

continued from last page			
		cvodes/cvodes_direct.h cvodes/cvodes_bandpre.h	cvodes/cvodes_spils.h cvodes/cvodes_bbdpre.h
ARKODE	Libraries	libsundials_arkode.*lib*	libsundials_farkode.a
	Header files	arkode/arkode.h arkode/arkode_direct.h arkode/arkode_bandpre.h	arkode/arkode_impl.h arkode/arkode_spils.h arkode/arkode_bbdpre.h
IDA	Libraries	libsundials_ida.*lib*	libsundials_fida.a
	Header files	ida/ida.h ida/ida_direct.h ida/ida_bbdpre.h	ida/ida_impl.h ida/ida_spils.h
IDAS	Libraries	libsundials_idas.*lib*	
	Header files	idas/idas.h idas/idas_direct.h idas/idas_bbdpre.h	idas/idas_impl.h idas/idas_spils.h
KINSOL	Libraries	libsundials_kinsol.*lib*	libsundials_fkinsol.a
	Header files	kinsol/kinsol.h kinsol/kinsol_direct.h kinsol/kinsol_bbdpre.h	kinsol/kinsol_impl.h kinsol/kinsol_spils.h

Appendix B

KINSOL Constants

Below we list all input and output constants used by the main solver and linear solver modules, together with their numerical values and a short description of their meaning.

B.1 KINSOL input constants

KINSOL **main solver module**		
KIN_ETACHOICE1	1	Use Eisenstat and Walker Choice 1 for η.
KIN_ETACHOICE2	2	Use Eisenstat and Walker Choice 2 for η.
KIN_ETACONSTANT	3	Use constant value for η.
KIN_NONE	0	Use inexact Newton globalization.
KIN_LINESEARCH	1	Use linesearch globalization.
Iterative linear solver module		
PREC_NONE	0	No preconditioning
PREC_RIGHT	2	Preconditioning on the right.
MODIFIED_GS	1	Use modified Gram-Schmidt procedure.
CLASSICAL_GS	2	Use classical Gram-Schmidt procedure.

B.2 KINSOL output constants

KINSOL **main solver module**		
KIN_SUCCESS	0	Successful function return.
KIN_INITIAL_GUESS_OK	1	The initial user-supplied guess already satisfies the stopping criterion.
KIN_STEP_LT_STPTOL	2	The stopping tolerance on scaled step length was satisfied.
KIN_WARNING	99	A non-fatal warning. The solver will continue.
KIN_MEM_NULL	-1	The kin_mem argument was NULL.
KIN_ILL_INPUT	-2	One of the function inputs is illegal.
KIN_NO_MALLOC	-3	The KINSOL memory was not allocated by a call to KINMalloc.

KIN_MEM_FAIL	-4	A memory allocation failed.
KIN_LINESEARCH_NONCONV	-5	The linesearch algorithm was unable to find an iterate sufficiently distinct from the current iterate.
KIN_MAXITER_REACHED	-6	The maximum number of nonlinear iterations has been reached.
KIN_MXNEWT_5X_EXCEEDED	-7	Five consecutive steps have been taken that satisfy a scaled step length test.
KIN_LINESEARCH_BCFAIL	-8	The linesearch algorithm was unable to satisfy the β-condition for `nbcfails` iterations.
KIN_LINSOLV_NO_RECOVERY	-9	The user-supplied routine preconditioner slve function failed recoverably, but the preconditioner is already current.
KIN_LINIT_FAIL	-10	The linear solver's initialization function failed.
KIN_LSETUP_FAIL	-11	The linear solver's setup function failed in an unrecoverable manner.
KIN_LSOLVE_FAIL	-12	The linear solver's solve function failed in an unrecoverable manner.
KIN_SYSFUNC_FAIL	-13	The system function failed in an unrecoverable manner.
KIN_FIRST_SYSFUNC_ERR	-14	The system function failed recoverably at the first call.
KIN_REPTD_SYSFUNC_ERR	-15	The system function had repeated recoverable errors.

KINDLS **linear solver module**

KINDLS_SUCCESS	0	Successful function return.
KINDLS_MEM_NULL	-1	The `kin_mem` argument was `NULL`.
KINDLS_LMEM_NULL	-2	The KINDLS linear solver has not been initialized.
KINDLS_ILL_INPUT	-3	The KINDLS solver is not compatible with the current NVECTOR module.
KINDLS_MEM_FAIL	-4	A memory allocation request failed.
KINDLS_JACFUNC_UNRECVR	-5	The Jacobian function failed in an unrecoverable manner.
KINDLS_JACFUNC_RECVR	-6	The Jacobian function had a recoverable error.
KINDLS_SUNMAT_FAIL	-7	An error occurred with the current SUNMATRIX module.

KINSPILS **linear solver modules**

KINSPILS_SUCCESS	0	Successful function return.
KINSPILS_MEM_NULL	-1	The `kin_mem` argument was `NULL`.
KINSPILS_LMEM_NULL	-2	The KINSPILS linear solver has not been initialized.
KINSPILS_ILL_INPUT	-3	The KINSPILS solver is not compatible with the current NVECTOR module, or an input value was illegal.
KINSPILS_MEM_FAIL	-4	A memory allocation request failed.
KINSPILS_PMEM_NULL	-5	The preconditioner module has not been initialized.
KINSPILS_SUNLS_FAIL	-6	An error occurred with the current SUNLINSOL module.

Bibliography

[1] KLU Sparse Matrix Factorization Library. http://faculty.cse.tamu.edu/davis/suitesparse.html.

[2] SuperLU_MT Threaded Sparse Matrix Factorization Library. http://crd-legacy.lbl.gov/ xiaoye/-SuperLU/.

[3] D. G. Anderson. Iterative procedures for nonlinear integral equations. *J. Assoc. Comput. Machinery*, 12:547–560, 1965.

[4] P. N. Brown. A local convergence theory for combined inexact-Newton/finite difference projection methods. *SIAM J. Numer. Anal.*, 24(2):407–434, 1987.

[5] P. N. Brown and A. C. Hindmarsh. Reduced Storage Matrix Methods in Stiff ODE Systems. *J. Appl. Math. & Comp.*, 31:49–91, 1989.

[6] P. N. Brown and Y. Saad. Hybrid Krylov Methods for Nonlinear Systems of Equations. *SIAM J. Sci. Stat. Comput.*, 11:450–481, 1990.

[7] G. D. Byrne. Pragmatic Experiments with Krylov Methods in the Stiff ODE Setting. In J.R. Cash and I. Gladwell, editors, *Computational Ordinary Differential Equations*, pages 323–356, Oxford, 1992. Oxford University Press.

[8] A. M. Collier and R. Serban. Example Programs for KINSOL v3.0.0. Technical Report UCRL-SM-208114, LLNL, 2017.

[9] T. A. Davis and P. N. Ekanathan. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Softw.*, 37(3), 2010.

[10] R. S. Dembo, S. C. Eisenstat, and T. Steihaug. Inexact Newton Methods. *SIAM J. Numer. Anal.*, 19:400–408, 1982.

[11] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.

[12] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM, Philadelphia, 1996.

[13] S. C. Eisenstat and H. F. Walker. Choosing the Forcing Terms in an Inexact Newton Method. *SIAM J. Sci. Comput.*, 17:16–32, 1996.

[14] H. Fang and Y. Saad. Two classes of secant methods for nonlinear acceleration. *Numer. Linear Algebra Appl.*, 16:197–221, 2009.

[15] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14:470–482, 1993.

[16] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *J. Research of the National Bureau of Standards*, 49(6):409–436, 1952.

[17] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS, suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, (31):363–396, 2005.

[18] A. C. Hindmarsh, R. Serban, and A. Collier. Example Programs for IDA v3.0.0. Technical Report UCRL-SM-208113, LLNL, 2017.

[19] A. C. Hindmarsh, R. Serban, and D. R. Reynolds. Example Programs for CVODE v3.0.0. Technical report, LLNL, 2017. UCRL-SM-208110.

[20] C. T. Kelley. *Iterative Methods for Solving Linear and Nonlinear Equations*. SIAM, Philadelphia, 1995.

[21] X. S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.

[22] P. A. Lott, H. F. Walker, C. S. Woodward, and U. M. Yang. An accelerated Picard method for nonlinear systems related to variably saturated flow. *Adv. Wat. Resour.*, 38:92–101, 2012.

[23] J. M. Ortega and W. C. Rheinbolt. *Iterative solution of nonlinear equations in several variables*. SIAM, Philadelphia, 2000. Originally published in 1970 by Academic Press.

[24] Daniel R. Reynolds. Example Programs for ARKODE v2.0.0. Technical report, Southern Methodist University, 2017.

[25] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14(2):461–469, 1993.

[26] Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 7:856–869, 1986.

[27] H. A. Van Der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 13:631–644, 1992.

[28] H. F. Walker and P. Ni. Anderson acceleration for fixed-point iterations. *SIAM Jour. Num. Anal.*, 49(4):1715–1735, 2011.

Index