DISCLAIMER

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

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

Approved for public release; further dissemination unlimited
Contents

List of Tables vii
List of Figures ix

1 Introduction 1
1.1 Historical Background ........................................... 1
1.2 Changes from previous versions ............................... 2
1.3 Reading this User Guide ........................................ 8
1.4 SUNDIALS Release License ..................................... 9
   1.4.1 Copyright Notices ....................................... 9
   1.4.1.1 SUNDIALS Copyright ................................ 10
   1.4.1.2 ARKode Copyright .................................... 10
1.4.2 BSD License ................................................ 10

2 Mathematical Considerations 13

3 Code Organization 21
3.1 SUNDIALS organization ....................................... 21
3.2 KINSOL organization .......................................... 21

4 Using KINSOL for C Applications 25
4.1 Access to library and header files .......................... 25
4.2 Data types ..................................................... 26
   4.2.1 Floating point types ................................... 26
   4.2.2 Integer types used for vector and matrix indices .... 26
4.3 Header files ................................................... 27
4.4 A skeleton of the user’s main program ...................... 28
4.5 User-callable functions ....................................... 30
   4.5.1 KINSOL initialization and deallocation functions .... 30
   4.5.2 Linear solver specification function .................. 31
   4.5.3 KINSOL solver function ................................ 33
   4.5.4 Optional input functions ................................ 34
      4.5.4.1 Main solver optional input functions .......... 36
      4.5.4.2 Linear solver interface optional input functions . 44
   4.5.5 Optional output functions ................................ 46
      4.5.5.1 SUNDIALS version information ................... 46
      4.5.5.2 Main solver optional output functions .......... 47
      4.5.5.3 KINS linear solver interface optional output functions . 49
4.6 User-supplied functions ....................................... 53
   4.6.1 Problem-defining function ............................... 53
   4.6.2 Error message handler function ......................... 53
   4.6.3 Informational message handler function ............... 54
   4.6.4 Jacobian construction (matrix-based linear solvers) .... 54
4.6.5 Jacobian-vector product (matrix-free linear solvers) ........................................ 56
4.6.6 Preconditioner solve (iterative linear solvers) .................................................... 56
4.6.7 Preconditioner setup (iterative linear solvers) ..................................................... 57
4.7 A parallel band-block-diagonal preconditioner module ........................................... 58

5 FKINSOL, an Interface Module for FORTRAN Applications ..................................... 63
5.1 Important note on portability .................................................................................... 63
5.2 Fortran Data Types .................................................................................................. 63
5.3 FKINSOL routines .................................................................................................... 64
5.4 Usage of the FKINSOL interface module ................................................................. 65
5.5 FKINSOL optional input and output ........................................................................ 71
5.6 Usage of the FKINBBD interface to KINBBDPRE .................................................... 71

6 Description of the NVECTOR module ........................................................................ 75
6.1 The NVECTOR_SERIAL implementation ................................................................. 84
6.2 The NVECTOR_PARALLEL implementation ......................................................... 86
6.3 The NVECTOR_OPENMP implementation ............................................................... 89
6.4 The NVECTOR_PTHREADS implementation ........................................................... 92
6.5 The NVECTOR_PARHYP implementation ............................................................... 94
6.6 The NVECTOR_PETSC implementation ................................................................... 96
6.7 The NVECTOR_CUDA implementation .................................................................... 97
6.8 The NVECTOR_RAJA implementation ..................................................................... 100
6.9 NVECTOR Examples .............................................................................................. 103
6.10 NVECTOR functions used by KINSOL ..................................................................... 106

7 Description of the SUNMatrix module ...................................................................... 109
7.1 The SUNMatrix_Dense implementation .................................................................... 112
7.2 The SUNMatrix_Band implementation ..................................................................... 115
7.3 The SUNMatrix_Sparse implementation ................................................................... 119
7.4 SUNMatrix Examples ............................................................................................ 125
7.5 SUNMatrix functions used by KINSOL ..................................................................... 126

8 Description of the SUNLinearSolver module ............................................................ 129
8.0.1 SUNLinearSolver core functions .......................................................................... 130
8.0.2 SUNLinearSolver set functions ............................................................................ 131
8.0.3 SUNLinearSolver get functions ............................................................................ 132
8.0.4 Functions provided by SUNDIALS packages ....................................................... 134
8.0.5 SUNLinearSolver return codes ............................................................................ 135
8.0.6 The generic SUNLinearSolver module ............................................................... 136
8.1 Compatibility of SUNLinearSolver modules ......................................................... 137
8.2 Implementing a custom SUNLinearSolver module ................................................. 137
8.3 The SUNLinearSolver_Dense implementation ....................................................... 138
8.3.1 SUNLINSOL_DENSE usage ............................................................................... 138
8.3.2 SUNLINSOL_DENSE description ...................................................................... 139
8.4 The SUNLinearSolver_Band implementation ....................................................... 140
8.4.1 SUNLINSOL_BAND usage ............................................................................... 140
8.4.2 SUNLINSOL_BAND description .................................................................... 141
8.5 The SUNLinearSolver_LapackDense implementation .......................................... 142
8.5.1 SUNLINSOL_LAPACKDENSE usage .................................................................. 142
8.5.2 SUNLINSOL_LAPACKDENSE description ....................................................... 143
8.6 The SUNLinearSolver_LapackBand implementation ............................................. 144
8.6.1 SUNLINSOL_LAPACKBAND usage .................................................................. 144
8.6.2 SUNLINSOL_LAPACKBAND description ....................................................... 145
8.7 The SUNLinearSolver_KLU implementation ......................................................... 146
8.7.1 SUNLINSOL_KLU usage ............................................................................... 146
## List of Tables

4.1 SUNDIALS linear solver interfaces and vector implementations that can be used for each.  30  
4.2 Optional inputs for KINSOL and KINLS  
4.3 Optional outputs from KINSOL and KINLS  
5.1 Keys for setting FKINSOL optional inputs  
5.2 Description of the FKINSOL optional output arrays IOUT and ROUT  
6.1 Vector Identifications associated with vector kernels supplied with SUNDIALS.  
6.2 Description of the NVVECTOR operations  
6.3 Description of the NVVECTOR fused operations  
6.4 Description of the NVVECTOR vector array operations  
6.5 List of vector functions usage by KINSOL code modules  
7.1 Identifiers associated with matrix kernels supplied with SUNDIALS.  
7.2 Description of the SUNMatrix operations  
7.3 SUNDIALS matrix interfaces and vector implementations that can be used for each.  
7.4 List of matrix functions usage by KINSOL code modules  
8.1 Description of the SUNLinearSolver error codes  
8.2 SUNDIALS matrix-based linear solvers and matrix implementations that can be used for each.  
8.3 List of linear solver functions usage by KINSOL code modules  
A.1 SUNDIALS libraries and header files  

VII
List of Figures

3.1 High-level diagram of the SUNDIALS suite .................................................. 22
3.2 Organization of the SUNDIALS suite .............................................................. 23
3.3 Overall structure diagram of the KINSOL package ......................................... 24

7.1 Diagram of the storage for a SUNMATRIX_BAND object ................................. 116
7.2 Diagram of the storage for a compressed-sparse-column matrix ....................... 122

A.1 Initial ccmake configuration screen .............................................................. 187
A.2 Changing the instdir ....................................................................................... 188
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 iterative 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 multiprocessor 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 v4.0.0-dev.2

KINSOL's previous direct and iterative linear solver interfaces, KINDLS and KINSPILS, have been merged into a single unified linear solver interface, KINLS, to support any valid SUNLINSOL module. The user interface for the new KINLS module is very similar to the previous KINDLS and KINSPILS interfaces; however to minimize challenges in user migration to the new names, the previous C and FORTRAN routine names may still be used; these will be deprecated in future releases, so we recommend that users migrate to the new names soon. We do note that FORTRAN users, however, may need to enlarge their iout array of optional integer outputs, and update the indices that they query for certain linear-solver-related statistics.

The names of all constructor routines for Sundials-provided SUNLINSOL implementations have been updated to SUNLinSolBand, SUNLinSol_Dense, SUNLinSol_KLU, SUNLinSol_LapackBand, SUNLinSol_LapackDense, SUNLinSol_PCG, SUNLinSol_SPTBCGS, SUNLinSol_SPTFQMR, SUNLinSol_SPFGMR, SUNLinSol_SPGMR, and SUNLinSol_SuperLU_MT. Solver-specific “set” routine names have been similarly standardized. To minimize challenges in user migration to the new names, the previous routine names may still be used; these will be deprecated in future releases, so we recommend that users migrate to the new names soon.

Changes in v4.0.0-dev.1

No changes were made to KINSOL in this release.

Changes in v4.0.0-dev

Three fused vector operations and seven vector array operations have been added to the NVECTOR API. These optional operations are intended to increase data reuse in vector operations, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. The new operations are NVLinearCombination, NVScaleAddMulti, NVDotProdMulti, NVLinearCombinationVectorArray, NVScaleVectorArray, NVConstVectorArray, NVWrmsNormVectorArray, NVWrmsNormMaskVectorArray, NVScaleAddMultiVectorArray, and NVLinearCombinationVectorArray. If any of these operations are defined as NULL in an NVECTOR implementation the NVECTOR interface will automatically call standard NVECTOR operations as necessary. Details on the new operations can be found in Chapter 6.
1.2 Changes from previous versions

Changes in v3.2.0

Fixed a problem with setting sunindextype which would occur with some compilers (e.g. armclang) that did not define __STDC_VERSION__.

Added hybrid MPI/CUDA and MPI/RAJA vectors to allow use of more than one MPI rank when using a GPU system. The vectors assume one GPU device per MPI rank.

Changed the name of the RAJA nvector library to libsundials_nveccudaraja.lib from libsundials_nvecraja.lib to better reflect that we only support CUDA as a backend for RAJA currently.

Several changes were made to the build system:

- CMake 3.1.3 is now the minimum required CMake version.
- Deprecate the behavior of the SUNDIALS_INDEX_TYPE CMake option and added the SUNDIALS_INDEX_SIZE CMake option to select the sunindextype integer size.
- The native CMake FindMPI module is now used to locate an MPI installation.
- If MPI is enabled and MPI compiler wrappers are not set, the build system will check if CMAKE_<language>_COMPILER can compile MPI programs before trying to locate and use an MPI installation.
- The previous options for setting MPI compiler wrappers and the executable for running MPI programs have been deprecated. The new options that align with those used in native CMake FindMPI module are MPI_C_COMPILER, MPI_CXX_COMPILER, MPI_Fortran_COMPILER, and MPIEXEC_EXECUTABLE.
- When a Fortran name-mangling scheme is needed (e.g., LAPACK_ENABLE is ON) the build system will infer the scheme from the Fortran compiler. If a Fortran compiler is not available or the inferred or default scheme needs to be overridden, the advanced options SUNDIALS_F77FUNC_CASE and SUNDIALS_F77FUNCUNDERSCORES can be used to manually set the name-mangling scheme and bypass trying to infer the scheme.
- Parts of the main CMakeLists.txt file were moved to new files in the src and example directories to make the CMake configuration file structure more modular.

Changes in v3.1.2

The changes in this minor release include the following:

- Updated the minimum required version of CMake to 2.8.12 and enabled using rpath by default to locate shared libraries on OSX.
- Fixed Windows specific problem where sunindextype was not correctly defined when using 64-bit integers for the sundials index type. On Windows sunindextype is now defined as the MSVC basic type __int64.
- Added sparse SUNMatrix “Reallocate” routine to allow specification of the nonzero storage.
- Updated the KLU SUNLinearSolver module to set constants for the two reinitialization types, and fixed a bug in the full reinitialization approach where the sparse SUNMatrix pointer would go out of scope on some architectures.
- Updated the “ScaleAdd” and “ScaleAddI” implementations in the sparse SUNMatrix module to more optimally handle the case where the target matrix contained sufficient storage for the sum, but had the wrong sparsity pattern. The sum now occurs in-place, by performing the sum
backwards in the existing storage. However, it is still more efficient if the user-supplied Jacobian routine allocates storage for the sum $I + \gamma J$ manually (with zero entries if needed).

- Changed the LICENSE install path to `instdir/include/sundials`.

### Changes in v3.1.1

The changes in this minor release include the following:

- Fixed a potential memory leak in the SPGMR and SPFGMR linear solvers: if “Initialize” was called multiple times then the solver memory was reallocated (without being freed).

- Updated KLU SUNLinearSolver module to use a `typedef` for the precision-specific solve function to be used (to avoid compiler warnings).

- Added missing typecasts for some `(void*)` pointers (again, to avoid compiler warnings).

- Bugfix in `summatrix_sparse.c` where we had used `int` instead of `sunindextype` in one location.

- Fixed a minor bug in `KINPrintInfo` where a case was missing for `KIN_REPTD_SYSFUNC_ERR` leading to an undefined info message.

- Added missing `#include <stdio.h>` in `nvector` and `sunmatrix` header files.

- Fixed an indexing bug in the CUDA NVECTOR implementation of `N_VWrmsNormMask` and revised the RAJA NVECTOR implementation of `N_VWrmsNormMask` to work with mask arrays using values other than zero or one. Replaced `double` with `realtype` in the RAJA vector test functions.

- Fixed compilation issue with GCC 7.3.0 and Fortran programs that do not require a SUNMATRIX or SUNLINSOL module (e.g., iterative linear solvers or fixed pointer solver).

In addition to the changes above, minor corrections were also made to the example programs, build system, and user documentation.

### Changes in v3.1.0

Added NVECTOR print functions that write vector data to a specified file (e.g., `N_VPrintFile_Serial`).

Added `make test` and `make test_install` options to the build system for testing SUNDIALS after building with `make` and installing with `make install` respectively.

### 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 DI and SI 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.
1.2 Changes from previous versions

- 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 the 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 fnmix name translation for Fkin_Spgmr.

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, `NGetVectorID`, that returns the `nvector` module name.

The Picard iteration return was changed 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 lambda 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 `limit` 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 `FKINIT` were added to split the `FKINMALLOC` routine into two pieces. `FKINMALLOC` remains for backward compatibility, but documentation for it has been removed.

A new example 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 `mmnewtstep` 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`, `SUNRpowerI`, 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 `FKINSETPIN`, the optional fourth argument `key_length` was removed, with hardcoded key string lengths passed to all `strncmp` tests.
1.2 Changes from previous versions

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, band-width 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

`KINSPEC`, `KINSPCSS`, `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 moni-
toring 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

UCRL-CODE-155951 (CVODE)
UCRL-CODE-155950 (CVODES)
UCRL-CODE-155952 (IDA)
UCRL-CODE-237203 (IDAS)
LLNL-CODE-665877 (KINSOL)
All rights reserved.

1.4.1.2 ARKode Copyright

ARKode is subject to the following joint Copyright notice. Copyright (c) 2015-2016, Southern Methodist University and Lawrence Livermore National Security Written by D.R. Reynolds, D.J. Gardner, A.C. Hindmarsh, C.S. Woodward, and J.M. Sexton.
LLNL-CODE-667205 (ARKODE)
All rights reserved.

1.4.2 BSD License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the disclaimer below.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the disclaimer (as noted below) in the documentation and/or other materials provided with the distribution.

3. Neither the name of the LLNS/LLNL nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL LAWRENCE LIVERMORE NATIONAL SECURITY, LLC, THE U.S. DEPARTMENT OF ENERGY OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Additional BSD Notice

1. This notice is required to be provided under our contract with the U.S. Department of Energy (DOE). This work was produced at Lawrence Livermore National Laboratory under Contract No. DE-AC52-07NA27344 with the DOE.

2. Neither the United States Government nor Lawrence Livermore National Security, LLC nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights.
3. Also, reference herein to any specific commercial products, process, or services by trade name, trademark, manufacturer or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.
Chapter 2
Mathematical Considerations

KINSOL solves nonlinear algebraic systems in real $N$-space.

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

$$F(u) = 0, \quad F : \mathbb{R}^N \rightarrow \mathbb{R}^N,$$  

(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

$$G(u) = u, \quad G : \mathbb{R}^N \rightarrow \mathbb{R}^N,$$  

(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 $n = 0, 1, 2, \ldots$ 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 $n$th 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 $\delta_n$, 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),
Mathematical Considerations

- 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 a direct linear solver, 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 solvers can only be used with the serial and threaded vector representations.

On the other hand, when using an iterative linear solver (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.

Additionally, it is possible for users to supply a matrix-based iterative linear solver to KINSOL, resulting in a Modified Inexact Newton method. As with the direct linear solvers, the Jacobian matrix is updated infrequently; similarly as with iterative linear solvers the linear system is solved only approximately.

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 matrix-based linear solvers, and 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),
- \(\text{mbset} = 10\) 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
- \(\|\lambda\delta_n\|_{D_u,\infty} < \text{steptol}\) with outdated Jacobian or preconditioner 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 \(\text{mbset}\), 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:

\[
\|z\|_{D_u} = \|D_u z\|_2, \quad \|z\|_{D_F} = \|D_F z\|_2, \quad \|z\|_{D_u,\infty} = \|D_u z\|_\infty, \quad \text{and} \quad \|z\|_{D_F,\infty} = \|D_F z\|_\infty
\]

(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 \( \delta_n \) 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 \( \lambda \) always set to 1. The other method is a global strategy, which attempts to use the direction implied by \( \delta_n \) 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 \( \alpha \) and \( \beta \) conditions of the Goldstein-Armijo linesearch given in [12] for step 2(b), where \( \lambda \) 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 \( \lambda \) such that \( u_n + \lambda \delta_n \) satisfies the sufficient decrease condition (or \( \alpha \)-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 \( \lambda \) to satisfy the so-called \( \beta \)-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, \( \lambda \) is allowed to vary in the interval \([\lambda_{\min}, \lambda_{\max}]\) where

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

and \( \lambda_{\max} \) 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 \( j \)th 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 $i$th 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 $\delta_n$ in Step 2(b) is scaled to take a step of that length.

Residual monitoring for Modified Newton method

When using a matrix-based linear solver, 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 $\text{mbsetsub} = 5$ 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 $\omega$ is given by

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

with $\rho$ defined as

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

where FTOL is the input scalar tolerance discussed before. Optionally, a constant value $\omega_{\text{const}}$ can be used for the parameter $\omega$.

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

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 $\eta_n$ 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 $\gamma$ and $\alpha$ are 0.9 and 2, respectively.

**Constant $\eta$**

$$
\eta_n = \text{constant},
$$

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 $\eta$ does not decrease too quickly [13].
Difference quotient Jacobian approximations

With the dense and banded matrix-based linear solvers, 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

\[ J_{ij} = \frac{F_i(u + \sigma_j e_j) - F_i(u)}{\sigma_j} \quad \text{(2.6)} \]

The increments \( \sigma_j \) are given by

\[ \sigma_j = \sqrt{U} \max \{|u^j|, 1/D^j_u\} \quad \text{(2.7)} \]

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, \texttt{relfunc}.

We note that with sparse and user-supplied matrix-based linear solvers, the Jacobian must be supplied by a user routine, i.e. it is not approximated internally within KINSOL.

In the case of a matrix-free iterative linear solver, 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

\[ J(u)v \approx \frac{F(u + \sigma v) - F(u)}{\sigma} \quad \text{(2.8)} \]

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

\[ \sigma = \max\{|u^T v|, u_{typ}^T |v|\} \frac{\|v\|_2}{\|v\|_2} \text{sign}(u^T v) \sqrt{U} \quad \text{(2.9)} \]

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 \textit{scaled} vectors \( u \) and \( v \), and so is applied to \( D_u u \) and \( D_v v \). The parameter \( U \) above can (optionally) be replaced by a user-specified value, \texttt{relfunc}. Convergence of the Newton method is maintained as long as the value of \( \sigma \) 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 \( n \)th 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 $n$th 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 $n = 0, 1, 2, \ldots$ until convergence do:
   
   (a) Set $m_n = \min \{m, n\}$
   (b) Set $F_n = (f_{n-m}, \ldots, f_n)$, where $f_i = G(u_i) - u_i$
   (c) Determine $\alpha^{(n)} = (\alpha_0^{(n)}, \ldots, \alpha_{m_n-1}^{(n)})$ 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 $n = 0, 1, 2, \ldots$ until convergence do:
   
   (a) Set $m_n = \min \{m, n\}$
   (b) Set $\Delta F_n = (\Delta f_{n-m}, \ldots, \Delta f_{n-1})$, 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

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

where $DF$ is a user-defined diagonal matrix that can be the identity or a scaling matrix chosen so that the components of $DF(G(u) - u)$ 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 Figs. 3.1 and 3.2). 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 $Mdy/dt = f_E(t, y) + f_I(t, y)$ based on additive Runge-Kutta methods;
- IDA, a solver for differential-algebraic systems $F(t, y, y') = 0$ 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 3.3. 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 solver modules is specified, and is then invoked as needed.

KINSOL now has a single unified linear solver interface, KINLS, supporting both direct and iterative linear solvers built using the generic SUNLINSOL API (see Chapter 8). These solvers may utilize a SUNMATRIX object (see Chapter 7) for storing Jacobian information, or they may be matrix-free. Since KINSOL can operate on any valid SUNLINSOL implementation, the set of linear solver modules available to KINSOL will expand as new SUNLINSOL modules are developed.

For users employing dense or banded Jacobian matrices, KINLS includes algorithms for their approximation 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 or user-supplied Jacobian matrices.

For users employing matrix-free iterative linear solvers, KINLS includes an algorithm for the approximation by difference quotients of the product between the Jacobian matrix and a vector, $Jv$. 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 KINSL, offer considerable assistance in building preconditioners.

KINSL’s linear solver interface consists of four primary phases, 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 KINSL module to each of the associated functions is fixed, thus allowing the central module to be completely independent of the linear system method.

KINSL also provides a preconditioner module called KINBBDPRE for use with any of the Krylov iterative linear solvers. It works in conjunction with NVVECTOR_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 KINSL 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 KINSL 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 KINSL memory structure. The reentrancy of KINSL was motivated by the anticipated multicomputer extension, but is also essential in a uniprocessor setting where two or more problems are solved by intermixed calls to the package from within a single user program.
3.2 KINSOL organization

Figure 3.2: Organization of the SUNDIALS suite
Figure 3.3: 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.
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 compatibility 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 compatibility 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/nvector`
- `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,

```c
#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. This includes the header file for kINLS, `kinsol/kinsol_ls.h`.

`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.

If using a Newton or Picard nonlinear solver that requires the solution of a linear system, then a linear solver module header file will be required. The header files corresponding to the various linear solver modules available for use with KINSOL are:

- Direct linear solvers:
  - `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;

- Iterative linear solvers:
  - `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 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 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_kryp` 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 $N_{\text{local}}$.

   Note: The variables $N$ and $N_{\text{local}}$ 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 $u = \text{N}_{\text{VMake}}(\ldots, \text{udata})$ 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 $u = \text{N}_{\text{VNew}}(\ldots)$, and then set its elements by accessing the underlying data with a call of the form $\text{ydata} = \text{N}_{\text{VGetArrayPointer}}(u)$. 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 $u = \text{N}_{\text{VMake}}(\ldots, \text{uvec})$, where `uvec` is a `hypre` or PETSc vector. Note that calls like $\text{N}_{\text{VNew}}(\ldots)$ and $\text{N}_{\text{VGetArrayPointer}}(\ldots)$ 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 $u = \text{N}_{\text{VMake}}(\ldots, c)$ 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 $u = \text{N}_{\text{VNew}}(\ldots)$, and then set its elements by accessing the underlying data where it is located with a call of the form $\text{N}_{\text{VGetDeviceArrayPointer}}(\ldots)$ or $\text{N}_{\text{VGetHostArrayPointer}}(\ldots)$. 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 $\text{kin\_mem} = \text{KINCreate}()$ to create the KINSOL memory block. $\text{KINCreate}$ returns a pointer to the KINSOL memory structure. See §4.5.1 for details.

5. **Allocate internal memory**

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

6. **Create matrix object**
If a matrix-based 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.

For the SUNDIALS-supplied SUNMATRIX implementations, the matrix object may be created using a call of the form

\[
\text{SUNMatrix } J = \text{SUNBandMatrix}(...);
\]

or

\[
\text{SUNMatrix } J = \text{SUNDenseMatrix}(...);
\]

or

\[
\text{SUNMatrix } J = \text{SUNSparseMatrix}(...);
\]

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

7. **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.

For any of the SUNDIALS-supplied SUNLINSOL implementations, the linear solver object may be created using a call of the form

\[
\text{SUNLinearSolver } LS = \text{SUNLinSol}(...);
\]

where * can be replaced with “Dense”, “SPGMR”, or other options, as discussed in §4.5.2 and Chapter 8.

8. **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.

9. **Attach linear solver module**

If a Newton or Picard iteration is chosen, initialize the KINLS 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):

\[
\text{ier } = \text{KINSetLinearSolver}(...);
\]

10. **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.

11. **Solve problem**

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

12. **Get optional outputs**

Call KINGet* functions to obtain optional output. See §4.5.5 for details.

13. **Deallocation 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:

\[
\text{N_VDestroy(u)};
\]

14. **Free solver memory**
Call \texttt{KINFree(\&kin\_mem)} to free the memory allocated for \texttt{KINSOL}.

15. Free linear solver and matrix memory

Call \texttt{SUNLinSolFree} and \texttt{SUNMatDestroy} to free any memory allocated for the linear solver and matrix objects created above.

16. Finalize MPI, if used

Call \texttt{MPI\_Finalize()} to terminate MPI.

\texttt{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 \texttt{SUNLINSOL} modules and the vector implementations required for use. As an example, one cannot use the dense direct solver interfaces with the \texttt{MPI}-based vector implementation. However, as discussed in Chapter 8 the \texttt{SUNDIALS} packages operate on generic \texttt{SUNLINSOL} objects, allowing a user to develop their own solvers should they so desire.

<table>
<thead>
<tr>
<th>Linear Solver</th>
<th>Serial</th>
<th>Parallel (MPI)</th>
<th>OpenMP</th>
<th>pThreads</th>
<th>hypre</th>
<th>PETSc</th>
<th>CUDA</th>
<th>RAJA</th>
<th>User Supp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Band</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>LapackDense</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>LapackBand</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>KLU</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>SUPERLUMT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>SPGMR</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>SPFGMR</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>SPBCGS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>SPTFQMR</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>PCG</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>User Supp.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

4.5 User-callable functions

This section describes the \texttt{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 \texttt{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 \texttt{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 function

As previously explained, Newton and Picard iterations require the solution of linear systems of the form $J\delta = -F$. Solution of these linear systems is handled using the KINLS linear solver interface. This interface supports all valid SUNLINSOL modules. Here, matrix-based SUNLINSOL modules utilize SUNMATRIX objects to store the Jacobian matrix $J = \partial F/\partial u$ and factorizations used throughout the solution process. Conversely, matrix-free SUNLINSOL modules instead use iterative methods to solve the linear systems of equations, and only require the action of the Jacobian on a vector, $Jv$.

With most iterative linear solvers, preconditioning can be done on the left only, on the right only, or not at all. However, only right preconditioning is supported within KINLS. If preconditioning is done, user-supplied functions define the linear operator corresponding to a 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 the function KINSetLinearSolver, as documented below. To create the SUNLinearSolver object, the user may call one of the SUNDIALS-packaged SUNLINSOL module constructor routines via a call of the form

```
SUNLinearSolver LS = SUNLinSol_*(...);
```

The current list of such constructor routines includes SUNLinSol_Dense, SUNLinSol_Band, SUNLinSol_LapackDense, SUNLinSol_LapackBand, SUNLinSol_KLU, SUNLinSol_SuperLUMT, SUNLinSol_SPGMR, SUNLinSol_SPFGMR, SUNLinSol_SPBCGS, SUNLinSol_SPTFQMR, and SUNLinSol_PCG.

Alternately, a user-supplied SUNLinearSolver module may be created and used instead. The use of each of the generic linear solvers involves certain constants, functions 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.

Once this solver object has been constructed, the user should attach it to KINSOL via a call to KINSetLinearSolver. The first argument passed to this function is the KINSOL memory pointer returned by KINCreate; the second argument is the desired SUNLINSOL object to use for solving Newton or Picard systems. The third argument is an optional SUNMATRIX object to accompany matrix-based SUNLINSOL inputs (for matrix-free linear solvers, the third argument should be NULL). A call to this function initializes the KINLS linear solver interface, linking it to the main KINSOL solver, and allows the user to specify additional parameters and routines pertinent to their choice of linear solver.

```
KINSetLinearSolver
```

**Call**

```
flag = KINSetLinearSolver(kin_mem, LS, J);
```

**Description**

The function KINSetLinearSolver attaches a generic SUNLINSOL object LS and corresponding template Jacobian SUNMATRIX object J (if applicable) to KINSOL, initializing the KINLS linear solver interface.

**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 (or NULL if not applicable).

**Return value**

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

- KINLS_SUCCESS The KINLS initialization was successful.
- KINLS_MEM_NULL The `kin_mem` pointer is NULL.
- KINLS_ILL_INPUT The KINLS interface is not compatible with the LS or J input objects or is incompatible with the current NVECTOR module.
- KINLS_SUNLINSOL_FAIL A call to the LS object failed.
- KINLS_MEM_FAIL A memory allocation request failed.

**Notes**

If LS is a matrix-based linear solver, then the template Jacobian matrix J will be used in the solve process, so if additional storage is required within the SUNMATRIX object (e.g. for factorization of a banded matrix), ensure that the input object is allocated with sufficient size (see the documentation of the particular SUNMATRIX type in Chapter 7 for further information).

The previous routines KINDlsSetLinearSolver and KINSpilsSetLinearSolver are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to the new routine name soon.
### 4.5 User-callable functions

#### 4.5.3 KINSOL solver function

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

```c
KIN_Sol
```

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 one 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 \( D_u \cdot u \) (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 \( u = u_0 = u \) 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_IILL_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 $\text{scteptol}$ is too large.

**KIN\_MAXITER\_REACHED**

The maximum number of nonlinear iterations has been reached.

**KIN\_MAXNEWT\_5X\_EXCEEDED**

Five consecutive steps have been taken that satisfy the inequality $\|D_F F(u)\|_{L_2} > 0.99 \text{mxnewtstep}$, where $p$ denotes the current step and $\text{mxnewtstep}$ is a scalar upper bound on the scaled step length. Such a failure may mean that $\|D_F F(u)\|_{L_2}$ asymptotes from above to a positive value, or the real scalar $\text{mxnewtstep}$ is too small.

**KIN\_LINESEARCH\_BCFAIL**

The line search algorithm was unable to satisfy the “beta-condition” for $\text{MXNBCF} + 1$ nonlinear iterations (not necessarily consecutive), which may indicate the algorithm is making poor progress.

**KIN\_LINSOLV\_NO\_RECOVERY**

The user-supplied routine $\text{psolve}$ encountered a recoverable error, but the preconditioner is already current.

**KIN\_LINIT\_FAIL**

The $\text{kinls}$ initialization routine ($\text{linit}$) encountered an error.

**KIN\_LSETUP\_FAIL**

The $\text{kinls}$ setup routine ($\text{lsetup}$) encountered an error; e.g., the user-supplied routine $\text{pset}$ (used to set up the preconditioner data) encountered an unrecoverable error.

**KIN\_LSOLVE\_FAIL**

The $\text{kinls}$ solve routine ($\text{lsolve}$) encountered an error; e.g., the user-supplied routine $\text{psolve}$ (used to solve the preconditioned linear system) encountered an unrecoverable error.

**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 $u\_\text{scale}$ and $f\_\text{scale}$ 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 $\text{flag} < 0$ will trap all $\text{KINSol}$ failures.

### 4.5.4 Optional input functions

There are numerous optional input parameters that control the behavior of the $\text{KINSol}$ solver. $\text{KINSol}$ provides functions that can be used to change these from their default values. Table 4.2 lists all optional input functions in $\text{KINSol}$ which are then described in detail in the remainder of this section, beginning with those for the main $\text{KINSol}$ solver and continuing with those for the $\text{KINLS}$ linear solver interface. For the most casual use of $\text{KINSol}$, 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 $\text{flag} < 0$ will catch any error.
## 4.5 User-callable functions

Table 4.2: Optional inputs for KINSOL and KINLS

<table>
<thead>
<tr>
<th>Optional input</th>
<th>Function name</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>KINSOL main solver</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error handler function</td>
<td>KINSetErrHandlerFn</td>
<td>internal fn.</td>
</tr>
<tr>
<td>Pointer to an error file</td>
<td>KINSetErrFile</td>
<td>stderr</td>
</tr>
<tr>
<td>Info handler function</td>
<td>KINSetInfoHandlerFn</td>
<td>internal fn.</td>
</tr>
<tr>
<td>Pointer to an info file</td>
<td>KINSetInfoFile</td>
<td>stdout</td>
</tr>
<tr>
<td>Data for problem-defining function</td>
<td>KINSetUserData</td>
<td>NULL</td>
</tr>
<tr>
<td>Verbosity level of output</td>
<td>KINSetPrintLevel</td>
<td>0</td>
</tr>
<tr>
<td>Max. number of nonlinear iterations</td>
<td>KINSetNumMaxIters</td>
<td>200</td>
</tr>
<tr>
<td>No initial matrix setup</td>
<td>KINSetNoInitSetup</td>
<td>SUNFALSE</td>
</tr>
<tr>
<td>No residual monitoring*</td>
<td>KINSetNoResMon</td>
<td>SUNFALSE</td>
</tr>
<tr>
<td>Max. iterations without matrix setup</td>
<td>KINSetMaxSetupCalls</td>
<td>10</td>
</tr>
<tr>
<td>Max. iterations without residual check*</td>
<td>KINSetMaxSubSetupCalls</td>
<td>5</td>
</tr>
<tr>
<td>Form of $\eta$ coefficient</td>
<td>KINSetEtaForm</td>
<td>KIN_ETACHOICE1</td>
</tr>
<tr>
<td>Constant value of $\eta$</td>
<td>KINSetEtaConstValue</td>
<td>0.1</td>
</tr>
<tr>
<td>Values of $\gamma$ and $\alpha$</td>
<td>KINSetEtaParams</td>
<td>0.9 and 2.0</td>
</tr>
<tr>
<td>Values of $\omega_{\text{min}}$ and $\omega_{\text{max}}$*</td>
<td>KINSetResMonParams</td>
<td>0.00001 and 0.9</td>
</tr>
<tr>
<td>Constant value of $\omega^*$</td>
<td>KINSetResMonConstValue</td>
<td>0.9</td>
</tr>
<tr>
<td>Lower bound on $\epsilon$</td>
<td>KINSetNoMinEps</td>
<td>SUNFALSE</td>
</tr>
<tr>
<td>Max. scaled length of Newton step</td>
<td>KINSetMaxNewtonStep</td>
<td>1000$|D_u u_0|_2$</td>
</tr>
<tr>
<td>Max. number of $\beta$-condition failures</td>
<td>KINSetMaxBetaFails</td>
<td>10</td>
</tr>
<tr>
<td>Rel. error for D.Q. $Jv$</td>
<td>KINSetRelErrFunc</td>
<td>$\sqrt{\text{uround}}$</td>
</tr>
<tr>
<td>Function-norm stopping tolerance</td>
<td>KINSetFuncNormTol</td>
<td>$\text{uround}^{1/3}$</td>
</tr>
<tr>
<td>Scaled-step stopping tolerance</td>
<td>KINSetScaledSteptol</td>
<td>$\text{uround}^{2/3}$</td>
</tr>
<tr>
<td>Inequality constraints on solution</td>
<td>KINSetConstraints</td>
<td>NULL</td>
</tr>
<tr>
<td>Nonlinear system function</td>
<td>KINSetSysFunc</td>
<td>none</td>
</tr>
<tr>
<td>Anderson Acceleration subspace size</td>
<td>KINSetMAA</td>
<td>0</td>
</tr>
<tr>
<td>KINLS linear solver interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jacobian function</td>
<td>KINSetJacFn</td>
<td>DQ</td>
</tr>
<tr>
<td>Preconditioner functions and data</td>
<td>KINSetPreconditioner</td>
<td>NULL, NULL, NULL</td>
</tr>
<tr>
<td>Jacobian-times-vector function and data</td>
<td>KINSetJacTimesVecFn</td>
<td>internal DQ, NULL</td>
</tr>
</tbody>
</table>
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.
4.5 User-callable functions

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 $\ell_2$ 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 iterative linear solver modules.
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 module.

**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.
4.5 User-callable functions

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 matrix-based linear solver modules.

**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 matrix-based linear solver modules.

**KINSetEtaForm**

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

Description The function KINSetEtaForm specifies the method for computing the value of the \( \eta \) 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 \( \eta \). 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.
Using KINSOL for C Applications

**KINSetEtaConstValue**

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

Description: The function `KINSetEtaConstValue` specifies the constant value for \( \eta \) in the case `etachoice = KIN_ETACONSTANT`.

Arguments:
- `kin_mem` (void *) pointer to the KINSOL memory block.
- `eta` (realtype) constant value for \( \eta \). 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 \( 0.0 < \eta \leq 1.0 \).

**KINSetEtaParams**

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

Description: The function `KINSetEtaParams` specifies the parameters \( \gamma \) and \( \alpha \) in the formula for \( \eta \), in the case `etachoice = KIN_ETACHOICE2`.

Arguments:
- `kin_mem` (void *) pointer to the KINSOL memory block.
- `egamma` (realtype) value of the \( \gamma \) parameter. Pass 0.0 to indicate the default.
- `ealpha` (realtype) value of the \( \alpha \) 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 \( 0.0 < \text{egamma} \leq 1.0 \) and \( 1.0 < \text{ealpha} \leq 2.0 \).

**KINSetResMonConstValue**

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

Description: The function `KINSetResMonConstValue` specifies the constant value for \( \omega \) when using residual monitoring.

Arguments:
- `kin_mem` (void *) pointer to the KINSOL memory block.
- `omegaconst` (realtype) constant value for \( \omega \). 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 \( \omega_{\text{min}} \) and \( \omega_{\text{max}} \) in the formula (2.4) for \( \omega \).

Arguments:
- `kin_mem` (void *) pointer to the KINSOL memory block.
- `omegamin` (realtype) value of the \( \omega_{\text{min}} \) parameter. Pass 0.0 to indicate the default.
4.5 User-callable functions

omegamax (realtype) value of the $\omega_{max}$ 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 omegamn or omegamax had an illegal value.

Notes The default values for omegamn and omegamax are 0.00001 and 0.9, respectively. The legal values are $0 < \text{omegamn} < \text{omegamax} < 1.0$.

KINSetNoMinEps

Call flag = KINSetNoMinEps(kin_mem, noMinEps);

Description The function KINSetNoMinEps specifies a flag that controls whether or not the value of $\epsilon$, 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 $\epsilon$.

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 $0.01^*\text{fnormtol}$, is applied to $\epsilon$. (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 ($\geq 0.0$). 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 $\|u_0\|_{D,\alpha}$, where $u_0$ is the initial guess.

KINSetMaxBetaFails

Call flag = KINSetMaxBetaFails(kin_mem, mxnbcf);

Description The function KINSetMaxBetaFails specifies the maximum number of $\beta$-condition failures in the linesearch algorithm.

Arguments
- kin_mem (void *) pointer to the KINSOL memory block.
- mxnbcf (realtype) maximum number of $\beta$-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.
Using KINSOL for C Applications

**KINSetRelErrFunc**

Call

```c
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 $F(u)$ ($relfunc \geq 0$). 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 $U = \text{unit roundoff}$.

---

**KINSetFuncNormTol**

Call

```c
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 ($\geq 0$). 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 $(\text{unit roundoff})^{1/3}$.

---

**KINSetScaledStepTol**

Call

```c
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 ($\geq 0$). 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 $(\text{unit roundoff})^{2/3}$.

---

**KINSetConstraints**

Call

```c
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

- \texttt{kin\_mem} (void *) pointer to the KINSOL memory block.
- \texttt{constraints} (N\_Vector) vector of constraint flags. If \texttt{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 \texttt{flag} (of type \texttt{int}) is one of:
- \texttt{KIN\_SUCCESS} The optional value has been successfully set.
- \texttt{KIN\_MEM\_NULL} The \texttt{kin\_mem} pointer is NULL.
- \texttt{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

\[ \text{flag} = \text{KINSetSysFunc}(	ext{kin\_mem}, \text{func}); \]

Description

The function \text{KINSetSysFunc} specifies the user-provided function that evaluates the nonlinear system function \( F(u) \) or \( G(u) \).

Arguments

- \texttt{kin\_mem} (void *) pointer to the KINSOL memory block.
- \texttt{func} (KINSysFn) user-supplied function that evaluates \( F(u) \) (or \( G(u) \) for fixed-point iteration).

Return value

The return value \texttt{flag} (of type \texttt{int}) is one of:
- \texttt{KIN\_SUCCESS} The optional value has been successfully set.
- \texttt{KIN\_MEM\_NULL} The \texttt{kin\_mem} pointer is NULL.
- \texttt{KIN\_ILL\_INPUT} The argument \texttt{func} was NULL.

Notes

The nonlinear system function is initially specified through \text{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

\[ \text{flag} = \text{KINSetMAA}(	ext{kin\_mem}, \text{maa}); \]

Description

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

Arguments

- \texttt{kin\_mem} (void *) pointer to the KINSOL memory block.
- \texttt{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 \texttt{flag} (of type \texttt{int}) is one of:
- \texttt{KIN\_SUCCESS} The optional value has been successfully set.
- \texttt{KIN\_MEM\_NULL} The \texttt{kin\_mem} pointer is NULL.
- \texttt{KIN\_ILL\_INPUT} The argument \texttt{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 $\text{maa}$ is 0, indicating no acceleration. The value of $\text{maa}$ should always be less than $\text{mxiter}$.

This function MUST be called before calling $\text{KINInit}$.

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

### 4.5.4.2 Linear solver interface optional input functions

For matrix-based linear solver modules, the $\text{KINLS}$ solver interface needs a function to compute an approximation to the Jacobian matrix $J(u)$. This function must be of type $\text{KINLsJacFn}$. 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 $\text{KINLS}$ solver. To specify a user-supplied Jacobian function $\text{jac}$, $\text{KINLS}$ provides the function $\text{KINSetJacFn}$. The $\text{KINLS}$ interface passes the pointer $\text{user_data}$ 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 $\text{user_data}$ may be specified through $\text{KINSetUserData}$.

#### $\text{KINSetJacFn}$

Call

```c
flag = KINSetJacFn(ida_mem, jac);
```

Description The function $\text{KINSetJacFn}$ specifies the Jacobian approximation function to be used.

Arguments

- $\text{kin_mem}$ (void *) pointer to the KINSOL memory block.
- $\text{jac}$ (KINLsJacFn) user-defined Jacobian approximation function.

Return value The return value $\text{flag}$ (of type int) is one of

- $\text{KINLS\_SUCCESS}$ The optional value has been successfully set.
- $\text{KINLS\_MEM\_NULL}$ The $\text{kin_mem}$ pointer is NULL.
- $\text{KINLS\_LMEM\_NULL}$ The $\text{KINLS}$ linear solver interface has not been initialized.

Notes By default, $\text{KINLS}$ uses an internal difference quotient function for dense and band matrices. If NULL is passed to $\text{jac}$, this default function is used. An error will occur if no $\text{jac}$ is supplied when using a sparse or user-supplied matrix.

This function must be called after the $\text{KINLS}$ linear solver interface has been initialized through a call to $\text{KINSetLinearSolver}$.

The function type $\text{KINLsJacFn}$ is described in §4.6.4.

The previous routine $\text{KINDlsSetJacFn}$ is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

When using matrix-free linear solver modules, the $\text{KINLS}$ linear solver interface requires 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 $\text{KINLS}$ solver interface. A user-defined Jacobian-vector function must be of type $\text{KINLsJacTimesVecFn}$ and can be specified through a call to $\text{KINLsSetJacTimesVecFn}$ (see §4.6.5 for specification details).

The pointer $\text{user_data}$ received through $\text{KINSetUserData}$ (or a pointer to NULL if $\text{user_data}$ was not specified) is passed to the Jacobian-times-vector function $\text{jtimes}$ each time it is called. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied functions without using global data in the program.
4.5 User-callable functions

KINSetJacTimesVecFn

Call

flag = KINSetJacTimesVecFn(kin_mem, jtimes);

Description
The function KINSetJacTimesVecFn specifies the Jacobian-vector product function.

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

jtimes (KINLSJacTimesVecFn) user-defined Jacobian-vector product function.

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

- KINLS_SUCCESS The optional values have been successfully set.
- KINLS_MEM_NULL The kin_mem pointer is NULL.
- KINLS_LMEM_NULL The KINLS linear solver has not been initialized.
- KINLS_SUNLS_FAIL An error occurred when setting up the system matrix-times-vector routines in the SUNLINSOL object used by the KINLS interface.

Notes
The default is to use an internal difference quotient for jtimes. If NULL is passed as jtimes, this default is used.

This function must be called after the KINLS linear solver interface has been initialized through a call to KINSetLinearSolver.

The function type KINLSJacTimesVecFn is described in §4.6.5.

The previous routine KINSpilsSetJacTimesVecFn is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

When using an iterative linear solver, the user may supply a preconditioning operator to aid in solution of the system. This operator consists of two user-supplied functions, psetup and psolve, that are supplied to KINLS using the function KINSetPreconditioner. The psetup function supplied to this routine should handle evaluation and preprocessing of any Jacobian data needed by the user’s preconditioner solve function, psolve. Both of these functions are fully specified in §4.6. The user data pointer received through KINSetUserData (or a pointer to NULL if user data was not specified) is passed to the 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.

KINSetPreconditioner

Call

flag = KINSetPreconditioner(kin_mem, psetup, psolve);

Description
The function KINSetPreconditioner specifies the preconditioner setup and solve functions.

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

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

psolve (KINLSPrecSolveFn) user-defined preconditioner solve function.

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

- KINLS_SUCCESS The optional values have been successfully set.
- KINLS_MEM_NULL The kin_mem pointer is NULL.
- KINLS_LMEM_NULL The KINLS linear solver has not been initialized.
- KINLS_SUNLS_FAIL An error occurred when setting up preconditioning in the SUNLINSOL object used by the KINLS interface.

Notes
The default is NULL for both arguments (i.e., no preconditioning).

This function must be called after the KINLS linear solver interface has been initialized through a call to KINSetLinearSolver.

The function type KINLSPrecSolveFn is described in §4.6.6.
Table 4.3: Optional outputs from KINSOL and KINLS

<table>
<thead>
<tr>
<th>Optional output</th>
<th>Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of KINSOL real and integer workspaces</td>
<td>KINGetWorkSpace</td>
</tr>
<tr>
<td>Number of function evaluations</td>
<td>KINGetNumFuncEvals</td>
</tr>
<tr>
<td>Number of nonlinear iterations</td>
<td>KINGetNumNolinSolvIters</td>
</tr>
<tr>
<td>Number of β-condition failures</td>
<td>KINGetNumBetaCondFails</td>
</tr>
<tr>
<td>Number of backtrack operations</td>
<td>KINGetNumBacktrackOps</td>
</tr>
<tr>
<td>Scaled norm of F</td>
<td>KINGetFuncNorm</td>
</tr>
<tr>
<td>Scaled norm of the step</td>
<td>KINGetStepLength</td>
</tr>
<tr>
<td>Size of KINLS real and integer workspaces</td>
<td>KINGetLinWorkSpace</td>
</tr>
<tr>
<td>No. of Jacobian evaluations</td>
<td>KINGetNumJacEvals</td>
</tr>
<tr>
<td>No. of F calls for D.Q. Jacobian[-vector] evals</td>
<td>KINGetNumLinFuncEvals</td>
</tr>
<tr>
<td>No. of linear iterations</td>
<td>KINGetNumLinIters</td>
</tr>
<tr>
<td>No. of linear convergence failures</td>
<td>KINGetNumLinConvFails</td>
</tr>
<tr>
<td>No. of preconditioner evaluations</td>
<td>KINGetNumPrecEvals</td>
</tr>
<tr>
<td>No. of preconditioner solves</td>
<td>KINGetNumPrecSolves</td>
</tr>
<tr>
<td>No. of Jacobian-vector product evaluations</td>
<td>KINGetNumJtimesEvals</td>
</tr>
<tr>
<td>Last return from a KINLS function</td>
<td>KINGetLastLinFlag</td>
</tr>
<tr>
<td>Name of constant associated with a return flag</td>
<td>KINGetLinReturnFlagName</td>
</tr>
</tbody>
</table>

The function type KINLSPrecSetupFn is described in §4.6.7.

The previous routine KINSpilsSetPreconditioner is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

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 KINLS linear solver interface. 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.

<table>
<thead>
<tr>
<th>SUNDIALSGetVersion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
</tr>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Arguments</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Return value</td>
</tr>
</tbody>
</table>
4.5 User-callable functions

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**

```c
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**

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

**Description**  The function **KINGetNumBetaCondFails** returns the number of $\beta$-condition failures.

**Arguments**

- `kin_mem` (void *) pointer to the KINSOL memory block.
- `nbcfails` (long int) number of $\beta$-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**

```c
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**

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

**Description**  The function **KINGetFuncNorm** returns the scaled Euclidean $\ell_2$ norm of the nonlinear system function $F(u)$ evaluated at the current iterate.

**Arguments**

- `kin_mem` (void *) pointer to the KINSOL memory block.
- `fnorm` (realttype) 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.
4.5 User-callable functions

KINGetStepLength

Call flag = KINGetStepLength(kin_mem, &steplength);

Description The function KINGetStepLength returns the scaled Euclidean $\ell_2$ 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 KINLS linear solver interface optional output functions

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

KINGetLinWorkSpace

Call flag = KINGetLinWorkSpace(kin_mem, &lenrwLS, &leniwLS);

Description The function KINGetLinWorkSpace returns the KINLS 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 KINLS workspace.
- leniwLS (long int) the number of integer values in the KINLS workspace.

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

- KINLS_SUCCESS The optional output value has been successfully set.
- KINLS_MEM_NULL The kin_mem pointer is NULL.
- KINLS_LMEM_NULL The KINLS linear solver interface 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 KINLS is not included in this report.

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

The previous routines KINDlsGetWorkspace and KINSpilsGetWorkspace are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

KINGetNumJacEvals

Call flag = KINGetNumJacEvals(kin_mem, &njevals);

Description The function KINGetNumJacEvals returns the cumulative number of calls to the KINLS 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

- KINLS_SUCCESS The optional output value has been successfully set.
KINLS_MEM_NULL The kin_mem pointer is NULL.
KINLS_LMEM_NULL The KINLS linear solver interface has not been initialized.

Notes
The previous routine KINDlsGetNumJacEvals is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

CALLING KINGetNumLinFuncEvals

Call
flag = KINGetNumLinFuncEvals(kin_mem, &nfevalsLS);

Description
The function KINGetNumLinFuncEvals returns the number of calls to the user system function used to compute the difference quotient approximation to the Jacobian or to the Jacobian-vector product.

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:

KINLS_SUCCESS The optional output value has been successfully set.
KINLS_MEM_NULL The kin_mem pointer is NULL.
KINLS_LMEM_NULL The KINLS linear solver interface has not been initialized.

Notes
The value nfevalsLS is incremented only if one of the default internal difference quotient functions is used.

The previous routines KINDlsGetNumFuncEvals and KINSpilsGetNumFuncEvals are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

CALLING KINGetNumLinIters

Call
flag = KINGetNumLinIters(kin_mem, &nliters);

Description
The function KINGetNumLinIters 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:

KINLS_SUCCESS The optional output value has been successfully set.
KINLS_MEM_NULL The kin_mem pointer is NULL.
KINLS_LMEM_NULL The KINLS linear solver interface has not been initialized.

Notes
The previous routine KINSpilsGetNumLinIters is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

CALLING KINGetNumLinConvFails

Call
flag = KINGetNumLinConvFails(kin_mem, &nlcfails);

Description
The function KINGetNumLinConvFails 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:

KINLS_SUCCESS The optional output value has been successfully set.
4.5 User-callable functions

KINLS_MEM_NULL  The \texttt{kin\_mem} pointer is NULL.
KINLS_LMEM_NULL  The KINLS linear solver interface has not been initialized.

Notes  The previous routine \texttt{KINSpilsGetNumConvFails} is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

\textbf{KINGetNumPrecEvals}

Call  \quad \texttt{flag = KINGetNumPrecEvals(kin\_mem, &npevals);}
Description  The function \texttt{KINGetNumPrecEvals} returns the cumulative number of preconditioner evaluations, i.e., the number of calls made to \texttt{psetup}.
Arguments  
  \texttt{kin\_mem} (void *) pointer to the \texttt{kinsol} memory block.
  \texttt{npevals} (long int) the current number of calls to \texttt{psetup}.
Return value  The return value \texttt{flag} (of type \texttt{int}) is one of:
  KINLS\_SUCCESS   The optional output value has been successfully set.
  KINLS\_MEM\_NULL   The \texttt{kin\_mem} pointer is NULL.
  KINLS\_LMEM\_NULL   The KINLS linear solver interface has not been initialized.

Notes  The previous routine \texttt{KINSpilsGetNumPrecEvals} is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

\textbf{KINGetNumPrecSolves}

Call  \quad \texttt{flag = KINGetNumPrecSolves(kin\_mem, &npsolves);}
Description  The function \texttt{KINGetNumPrecSolves} returns the cumulative number made to the preconditioner solve function, \texttt{psolve}.
Arguments  
  \texttt{kin\_mem} (void *) pointer to the \texttt{kinsol} memory block.
  \texttt{npsolves} (long int) the current number of calls to \texttt{psolve}.
Return value  The return value \texttt{flag} (of type \texttt{int}) is one of:
  KINLS\_SUCCESS   The optional output value has been successfully set.
  KINLS\_MEM\_NULL   The \texttt{kin\_mem} pointer is NULL.
  KINLS\_LMEM\_NULL   The KINLS linear solver interface has not been initialized.

Notes  The previous routine \texttt{KINSpilsGetNumPrecSolves} is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

\textbf{KINGetNumJtimesEvals}

Call  \quad \texttt{flag = KINGetNumJtimesEvals(kin\_mem, &njvevals);}
Description  The function \texttt{KINGetNumJtimesEvals} returns the cumulative number made to the Jacobian-vector product function, \texttt{jtimes}.
Arguments  
  \texttt{kin\_mem} (void *) pointer to the \texttt{kinsol} memory block.
  \texttt{njvevals} (long int) the current number of calls to \texttt{jtimes}.
Return value  The return value \texttt{flag} (of type \texttt{int}) is one of:
  KINLS\_SUCCESS   The optional output value has been successfully set.
  KINLS\_MEM\_NULL   The \texttt{kin\_mem} pointer is NULL.
  KINLS\_LMEM\_NULL   The KINLS linear solver interface has not been initialized.
The previous routine KINSpilsGetNumJtimesEvals is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

KINGetLastLinFlag

Call
flag = KINGetLastLinFlag(kin_mem, &lsflag);

Description The function KINGetLastLinFlag returns the last return value from a KINLS routine.

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

Return value The return value flag (of type int) is one of
KINLS_SUCCESS The optional output value has been successfully set.
KINLS_MEM_NULL The kin_mem pointer is NULL.
KINLS_LMEM_NULL The KINLS linear solver interface has not been initialized.

Notes If the KINLS setup function failed (i.e. KINSolve returned KINLSSETUP_FAIL) when using the SUNLINSOL_DENSE or SUNLINSOL_BAND modules, 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.

If the KINLS setup function failed when using another SUNLINSOL module, then lsflag will be SUNLS_PSET_FAIL_UNREC, SUNLSASET_FAIL_UNREC, or SUNLS_PACKAGE_FAIL_UNREC.

If the KINLS solve function failed (i.e., KINSol returned KINLSOLVE_FAIL), then 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_PPSOLVE_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_QRSOLVE_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.

The previous routines KINDisGetLastFlag and KINSpilsGetLastFlag are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

KINGetLinReturnFlagName

Call
name = KINGetLinReturnFlagName(lsflag);

Description The function KINGetLinReturnFlagName returns the name of the KINLS constant corresponding to lsflag.

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

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

Notes The previous routines KINDisGetReturnFlagName and KINSpilsGetReturnFlagName are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to the new routine name soon.
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 provide 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:

```c
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 construction (matrix-based linear solvers)

If a matrix-based linear solver module is used (i.e., a non-NULL SUNMATRIX object J was supplied to KINSetLinearSolver), the user may provide a function of type KINLsJacFn defined as follows:

```c
KINLsJacFn
Definition typedef int (*KINLsJacFn)(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 KINLsJacFn 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 \( J(u) \) is zeroed out, so only nonzero elements need to be loaded into \( J \).
If the user’s KINLsJacFn 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.
4.6 User-supplied functions

**dense:**
A user-supplied dense Jacobian function must load the $N \times N$ dense matrix $J$ with an approximation to the Jacobian matrix $J(u)$ at the point $(u)$. The accessor macros $\text{SM\_ELEMENT\_D}$ and $\text{SM\_COLUMN\_D}$ allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the $\text{SUN\_MATRIX\_DENSE}$ type. $\text{SM\_ELEMENT\_D}(J, i, j)$ references the $(i, j)$-th element of the dense matrix $J$ (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 $\text{SM\_ELEMENT\_D}(J, m-1, n-1) = J_{m,n}$. Alternatively, $\text{SM\_COLUMN\_D}(J, j)$ returns a pointer to the first element of the $j$-th column of $J$ (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 $\text{col}_n = \text{SM\_COLUMN\_D}(J, n-1); \text{col}_n[m-1] = J_{m,n}$. For large problems, it is more efficient to use $\text{SM\_COLUMN\_D}$ than to use $\text{SM\_ELEMENT\_D}$. Note that both of these macros number rows and columns starting from 0. The $\text{SUN\_MATRIX\_DENSE}$ type and accessor macros are documented in §7.1.

**banded:**
A user-supplied banded Jacobian function must load the $N \times N$ banded matrix $J$ with an approximation to the Jacobian matrix $J(u)$ at the point $(u)$. The accessor macros $\text{SM\_ELEMENT\_B}$, $\text{SM\_COLUMN\_B}$, and $\text{SM\_COLUMN\_ELEMENT\_B}$ allow the user to read and write banded matrix elements without making specific references to the underlying representation of the $\text{SUN\_MATRIX\_BAND}$ type. $\text{SM\_ELEMENT\_B}(J, i, j)$ references the $(i, j)$-th element of the banded matrix $J$, 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 $\text{mupper}$ and $\text{mlower}$, the Jacobian element $J_{m,n}$ can be loaded using the statement $\text{SM\_ELEMENT\_B}(J, m-1, n-1) = J_{m,n}$. The elements within the band are those with $-\text{mupper} \leq m-n \leq \text{mlower}$. Alternatively, $\text{SM\_COLUMN\_B}(J, j)$ returns a pointer to the diagonal element of the $j$-th column of $J$, and if we assign this address to $\text{realtype *col}_j$, then the $i$-th element of the $j$-th column is given by $\text{SM\_COLUMN\_ELEMENT\_B}(\text{col}_j, i, j)$, counting from 0. Thus, for $(m,n)$ within the band, $J_{m,n}$ can be loaded by setting $\text{col}_n = \text{SM\_COLUMN\_B}(J, n-1)$; and $\text{SM\_COLUMN\_ELEMENT\_B}(\text{col}_n, m-1, n-1) = J_{m,n}$. 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 banded matrix of type $\text{SUN\_MATRIX\_BAND}$. The array $\text{col}_n$ can be indexed from $-\text{mupper}$ to $\text{mlower}$. For large problems, it is more efficient to use $\text{SM\_COLUMN\_B}$ and $\text{SM\_COLUMN\_ELEMENT\_B}$ than to use the $\text{SM\_ELEMENT\_B}$ macro. As in the dense case, these macros all number rows and columns starting from 0. The $\text{SUN\_MATRIX\_BAND}$ type and accessor macros are documented in §7.2.

**sparse:**
A user-supplied sparse Jacobian function must load the $N \times N$ compressed-sparse-column or compressed-sparse-row matrix $J$ with an approximation to the Jacobian matrix $J(u)$ at the point $(u)$. Storage for $J$ already exists on entry to this function, although the user should ensure that sufficient space is allocated in $J$ 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 $\text{SUN\_MATRIX\_SPARSE}$ object may be accessed using the macro $\text{SM\_NNZ\_S}$ or the routine $\text{SUN\_Sparse\_Matrix\_NNZ}$. The $\text{SUN\_MATRIX\_SPARSE}$ type and accessor macros are documented in §7.3.

The previous function type $\text{KIND\_lsJacFn}$ is identical to $\text{KINLsJacFn}$, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new function type name soon.
4.6.5 Jacobian-vector product (matrix-free linear solvers)

If a matrix-free linear solver is to be used (i.e., a NULL-valued SUNMATRIX was supplied to KINSetLinearSolver), the user may provide a function of type KINLsJacTimesVecFn 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.

**KINLsJacTimesVecFn**

definition typedef int (*KINLsJacTimesVecFn)(N_Vector v, N_Vector Jv, 
N_Vector u, boolantype 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.

This function must return a value of $J * v$ that uses the current value of $J$, i.e. as evaluated at the current $u$.

If the user’s KINLsJacTimesVecFn 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.

The previous function type KINSpilsJacTimesVecFn is identical to KINLsJacTimesVecFn, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new function type name soon.

4.6.6 Preconditioner solve (iterative linear solvers)

If a user-supplied preconditioner is to be used with a SUNLINSOL solver module, 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 KINLsPrecSolveFn, defined as follows:
4.6 User-supplied functions

**KINLsPrecSolveFn**

**Definition**
```c
typedef int (*KINLsPrecSolveFn)(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.

The previous function type `KINSpilsPrecSolveFn` is identical to `KINLsPrecSolveFn`, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new function type name soon.

### 4.6.7 Preconditioner setup (iterative linear solvers)

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 `KINLsPrecSetupFn`, defined as follows:

**KINLsPrecSetupFn**

**Definition**
```c
typedef int (*KINLsPrecSetupFn)(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 `KINLsPrecSetupFn` 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.

The previous function type `KINSpilsPrecSetupFn` is identical to `KINLsPrecSetupFn`, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new function type name soon.

### 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 Jacobian 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.

```c
typedef void (*KINBBDLocalFn)(sunindextype Nlocal, N_Vector u,
                               N_Vector gval, void *user_data);
```
4.7 A parallel band-block-diagonal preconditioner module

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 KINBBBDLocalFn 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. Allocate internal memory
6. Create linear solver object
   - When creating the iterative linear solver object, specify use of right preconditioning (PREC_RIGHT) as KINSOL only supports right preconditioning.
7. Attach linear solver module
8. **Initialize the KINBBDPRE preconditioner module**

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

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

to allocate memory for and initialize the internal preconditioner data. The last two arguments of \texttt{KINBBDPrecInit} are the two user-supplied functions described above.

9. Set optional inputs

Note that the user should not overwrite the preconditioner data, setup function, or solve function through calls to \texttt{KINSetPreconditioner} optional input functions.

10. Solve problem

11. **Get optional output**

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

12. Deallocate memory for solution vector

13. Free solver memory

14. Free linear solver memory

15. Finalize MPI, if used

The user-callable function that initializes \texttt{KINBBDPRE} (step 8), is described in more detail below.

```
KINBBDPrecInit
```

**Call**

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

**Description**
The function \texttt{KINBBDPrecInit} initializes and allocates memory for the \texttt{KINBBDPRE} preconditioner.

**Arguments**
- \texttt{kin_mem} (void *) pointer to the \texttt{KINSOL} memory block.
- \texttt{Nlocal} (sunindextype) local vector length.
- \texttt{mudq} (sunindextype) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
- \texttt{mldq} (sunindextype) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
- \texttt{mukeep} (sunindextype) upper half-bandwidth of the retained banded approximate Jacobian block.
- \texttt{mlkeep} (sunindextype) lower half-bandwidth of the retained banded approximate Jacobian block.
- \texttt{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 \(dq\_rel\_u = 0.0\).
- \texttt{Gloc} (KINBBDLocalFn) the C function which computes the approximation \(G(u) \approx F(u)\).
- \texttt{Gcomm} (KINBBDCommFn) the optional C function which performs all interprocess communication required for the computation of \(G(u)\).

**Return value**
The return value \texttt{flag} (of type \texttt{int}) is one of

- \texttt{KINLS_SUCCESS} The call to \texttt{KINBBDPrecInit} was successful.
KINLS_MEM_NULL  The kin_mem pointer was NULL.
KINLS_MEM_FAIL  A memory allocation request has failed.
KINLS_ILMEM_NULL  The KINLS linear solver interface has not been initialized.
KINLS_ILL_INPUT  The supplied vector implementation was not compatible with the block band preconditioner.

Notes  If one of the half-bandwidths mudq or mlldq 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 mlldq 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 KINBBBDPRE module:

### KINBBBDPrecGetWorkSpace

Call  
\[
\text{flag} = \text{KINBBBDPrecGetWorkSpace}(\text{kin}\_\text{mem}, &\text{lenrWBBDP}, &\text{leniwBBDP});
\]

Description  
The function KINBBBDPrecGetWorkSpace returns the local KINBBBDPRE 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 KINBBBDPRE workspace.
- **leniwBBDP**  (long int) local number of integer values in the KINBBBDPRE workspace.

Return value  
The return value **flag** (of type int) is one of:
- KINLS_SUCCESS  The optional output value has been successfully set.
- KINLS_MEM_NULL  The kin_mem pointer was NULL.
- KINLS_PMEM_NULL  The KINBBBDPRE preconditioner has not been initialized.

Notes  
The workspace requirements reported by this routine correspond only to memory allocated within the KINBBBDPRE 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 KINGetLinWorkSpace function.

### KINBBBDPrecGetNumGfnEvals

Call  
\[
\text{flag} = \text{KINBBBDPrecGetNumGfnEvals}(\text{kin}\_\text{mem}, &\text{ngevalsBBDP});
\]

Description  
The function KINBBBDPrecGetNumGfnEvals returns the number of calls to the user Gloc function due to the difference quotient approximation of the Jacobian blocks used within KINBBBDPRE’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:
- KINLS_SUCCESS  The optional output value has been successfully set.
- KINLS_MEM_NULL  The kin_mem pointer was NULL.
- KINLS_PMEM_NULL  The KINBBBDPRE preconditioner has not been initialized.

Notes  
In addition to the ngevalsBBDP Gloc evaluations, the costs associated with KINBBBDPRE 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 \S4.5.5).
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 $\text{F77\_FUNC}$ defined in the header file sundials_config.h. The mapping defined by $\text{F77\_FUNC}$ 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
• \texttt{int64\_t} – equivalent to an \texttt{INTEGER*8} in \texttt{FORTRAN}

In general, for the \texttt{FORTRAN} interfaces in \textsc{sundials}, flags of type \texttt{int}, vector and matrix lengths, counters, and arguments to \texttt{*SETIN()} functions all have \texttt{long int} type, and \texttt{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 \texttt{sunindextype}, they can include \texttt{sundials\_config.h}.

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

• \texttt{single} – equivalent to a \texttt{REAL} or \texttt{REAL*4} in \texttt{FORTRAN}
• \texttt{double} – equivalent to a \texttt{DOUBLE PRECISION} or \texttt{REAL*8} in \texttt{FORTRAN}
• \texttt{extended} – equivalent to a \texttt{REAL*16} in \texttt{FORTRAN}

5.3 \textsc{FKINSOL} routines

The user-callable functions, with the corresponding \textsc{kinsol} functions, are as follows:

• Interface to the \texttt{NVECTOR} modules
  
  – \texttt{FNVINITS} (defined by \texttt{NVECTOR\_SERIAL}) interfaces to \texttt{N\_VNewEmpty\_Serial}.
  
  – \texttt{FNVINITP} (defined by \texttt{NVECTOR\_PARALLEL}) interfaces to \texttt{N\_VNewEmpty\_Parallel}.
  
  – \texttt{FNVINITOMP} (defined by \texttt{NVECTOR\_OPENMP}) interfaces to \texttt{N\_VNewEmpty\_OpenMP}.
  
  – \texttt{FNVINITPTS} (defined by \texttt{NVECTOR\_PTHREADS}) interfaces to \texttt{N\_VNewEmpty\_Pthreads}.

• Interface to the \texttt{SUNMATRIX} modules

  – \texttt{FSUNBANDMATINIT} (defined by \texttt{SUNMATRIX\_BAND}) interfaces to \texttt{SUNBandMatrix}.
  
  – \texttt{FSUNDENSEMATINIT} (defined by \texttt{SUNMATRIX\_DENSE}) interfaces to \texttt{SUNDenseMatrix}.
  
  – \texttt{FSUNSPARSEMATINIT} (defined by \texttt{SUNMATRIX\_SPARSE}) interfaces to \texttt{SUNSparseMatrix}.

• Interface to the \texttt{SUNLINSOL} modules

  – \texttt{FSUNBANDLINSOLINIT} (defined by \texttt{SUNLINSOL\_BAND}) interfaces to \texttt{SUNLinSol\_Band}.
  
  – \texttt{FSUNDENSELINSOLINIT} (defined by \texttt{SUNLINSOL\_DENSE}) interfaces to \texttt{SUNLinSol\_Dense}.
  
  – \texttt{FSUNKLUINIT} (defined by \texttt{SUNLINSOL\_KLU}) interfaces to \texttt{SUNLinSol\_KLU}.
  
  – \texttt{FSUNKLUREINIT} (defined by \texttt{SUNLINSOL\_KLU}) interfaces to \texttt{SUNLinSol\_KLUReinit}.
  
  – \texttt{FSUNLAPACKBANDINIT} (defined by \texttt{SUNLINSOL\_LAPACKBAND}) interfaces to \texttt{SUNLinSol\_LapackBand}.
  
  – \texttt{FSUNLAPACKDENSEINIT} (defined by \texttt{SUNLINSOL\_LAPACKDENSE}) interfaces to \texttt{SUNLinSol\_LapackDense}.
  
  – \texttt{FSUNPGGINIT} (defined by \texttt{SUNLINSOL\_PGC}) interfaces to \texttt{SUNLinSol\_PCG}.
  
  – \texttt{FSUNSPBCGSINIT} (defined by \texttt{SUNLINSOL\_SPBCGS}) interfaces to \texttt{SUNLinSol\_SPBCGS}.
  
  – \texttt{FSUNSPFGMRINIT} (defined by \texttt{SUNLINSOL\_SPFGMR}) interfaces to \texttt{SUNLinSol\_SPFGMR}.
  
  – \texttt{FSUNSPGMRINIT} (defined by \texttt{SUNLINSOL\_SPGMR}) interfaces to \texttt{SUNLinSol\_SPGMR}.
  
  – \texttt{FSUNSTPFQMRINIT} (defined by \texttt{SUNLINSOL\_STPFQMR}) interfaces to \texttt{SUNLinSol\_STPFQMR}.
  
  – \texttt{FSUNSUPERLUMTINIT} (defined by \texttt{SUNLINSOL\_SUPERLUMT}) interfaces to \texttt{SUNLinSol\_SuperLUMT}.

• Interface to the main \textsc{kinsol} module

  – \texttt{FKINCREATE} interfaces to \texttt{KINCreate}. 
5.4 Usage of the FKINSOL interface module

- FKINSETIIN and FKINSETRIN interface to KINSet* functions.
- FKINIT 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 KINLS module
  - FKINLSINIT interfaces to KINSetLinearSolver.
  - FKINLSSETJAC interfaces to KINSetJacTimesVecFn.
  - FKINLSSETPREC interfaces to KINSetPreconditioner.
  - FKINDENSESETJAC interfaces to KINSetJacFn.
  - FKINBANDSETJAC interfaces to KINSetJacFn.
  - FKINSPARSESETJAC interfaces to KINSetJacFn.

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

<table>
<thead>
<tr>
<th>FKINSOL routine (Fortran, user-supplied)</th>
<th>KINSOL function (C, interface)</th>
<th>KINSOL type of interface function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FKFUN</td>
<td>FKINfunc</td>
<td>KINSysFn</td>
</tr>
<tr>
<td>FKDJAC</td>
<td>FKINDenseJac</td>
<td>KINLsJacFn</td>
</tr>
<tr>
<td>FKBJac</td>
<td>FKBandJac</td>
<td>KINLsJacFn</td>
</tr>
<tr>
<td>FKINSPJAC</td>
<td>FKINSparseJac</td>
<td>KINLsJacFn</td>
</tr>
<tr>
<td>FKPSET</td>
<td>FKPSet</td>
<td>KINLsPrecSetupFn</td>
</tr>
<tr>
<td>FKPSOL</td>
<td>FKINPSol</td>
<td>KINLsPrecSolveFn</td>
</tr>
<tr>
<td>FKJTIMES</td>
<td>FKNJtimes</td>
<td>KINLsJacTimesVecFn</td>
</tr>
</tbody>
</table>

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

   ```fortran
   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

\[
\text{CALL FNVINIT***(....)}
\]

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 a matrix-based *SUNLINSOL* linear solver module and one of the *sunmatrix* modules supplied with *SUNDIALS*, the user must make a call of the form

\[
\text{CALL FSUN***MATINIT(...)}
\]

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

\[
\begin{align*}
\text{CALL FSUNBANDLINSOLINIT(...)} \\
\text{CALL FSUNDENSELINSOLINIT(...)} \\
\text{CALL FSUNKLUINIT(...)} \\
\text{CALL FSUNLAPACKBANDINIT(...)} \\
\text{CALL FSUNLAPACKDENSEINIT(...)} \\
\text{CALL FSUNPCGINIT(...)} \\
\text{CALL FSUNPCGSETPRECTYPE(...)} \\
\text{CALL FSUNSPBCGSSETPRECTYPE(...)} \\
\text{CALL FSUNSPFGMRSETGSTYPE(...)} \\
\text{CALL FSUNSPGMRSETPRECTYPE(...)} \\
\text{CALL FSUNSPTFQMRSETGSTYPE(...)} \\
\text{CALL FSUNSPTFQMRSETPRECTYPE(...)} \\
\text{CALL FSUNSUPERLUMTINIT(...)}
\end{align*}
\]

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

\[
\begin{align*}
\text{CALL FSUNKLUSETORDERING(...)} \\
\text{CALL FSUNSUPERLUMTSETORDERING(...)} \\
\text{CALL FSUNPCGSETPRECTYPE(...)} \\
\text{CALL FSUNPCGSETPRECTYPE(...)} \\
\text{CALL FSUNSPBCGSSETPRECTYPE(...)} \\
\text{CALL FSUNSPFGMRSETGSTYPE(...)} \\
\text{CALL FSUNSPGMRSETGSTYPE(...)} \\
\text{CALL FSUNSPTFQMRSETGSTYPE(...)} \\
\text{CALL FSUNSPTFQMRSETPRECTYPE(...)} \\
\text{CALL FSUNSUPERLUMTSETMAXL(...)}
\end{align*}
\]

where again the call sequences are described in the appropriate sections of Chapter 8.
5.4 Usage of the FKINSOL interface module

5. **Problem specification**

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

```fortran
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:

```fortran
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 KINLS linear solver interface.

To attach any SUNLINSOL object (and optional SUNMATRIX object) to the KINLS interface, then following calls to initialize the SUNLINSOL (and SUNMATRIX) object(s) in steps 3 and 4 above, the user must make the call:

```fortran
CALL FKINLSINIT (IER)
```

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

The previous routines FKINDLSINIT and FKINSPILSINIT are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to the new routine name soon.
**KINLS with dense Jacobian matrix**

As an option when using the KINLS interface with the SUNLINSOL_DENSE or SUNLINSOL_LAPACKDENSE linear solvers, 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:

```fortran
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 \( FKINLSINIT \), the user must make the call:

```fortran
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.

**KINLS with band Jacobian matrix**

As an option when using the KINLS interface with the 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:

```fortran
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 \( FKINLSINIT \), the user must make the call:

```fortran
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.

**KINLS with sparse Jacobian matrix**

When using the KINLS interface with either of the SUNLINSOL_KLU or SUNLINSOL_SUPERLUMT linear solvers, the user must supply the \( FKINSPJAC \) routine that computes a compressed-sparse-column or compressed-sparse-row 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 \texttt{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 \texttt{FKINSPJAC} routine has been provided, then following the call to \texttt{FKINLSINIT}, the following call must be made

\texttt{CALL FKINSPARSESETJAC (IER)}

The int return flag \( IER \) is an error return flag which is 0 for success or nonzero for an error.

**KINLS with Jacobian-vector product**

As an option when using the \texttt{KINLS} linear solver interface, the user may supply a routine that computes the product of the system Jacobian and a given vector. If supplied, it must have the following form:

\begin{verbatim}
SUBROUTINE FKINJTIMES (V, FJV, NEWU, U, IER)
DIMENSION V(*), FJV(*), U(*)
\end{verbatim}

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 \texttt{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 \texttt{FKINJTIMES} should recompute any saved Jacobian data it uses and reset \( NEWU \) to 0. (See \S4.6.5.)

To indicate that the \texttt{FKINJTIMES} routine has been provided, then following the call to \texttt{FKINLSINIT}, the following call must be made

\texttt{CALL FKINLSSETJAC (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.

The previous routine \texttt{FKINPSILSETJAC} is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

**KINLS with preconditioning**

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

\begin{verbatim}
SUBROUTINE FKPSOL (U, USCALE, FVAL, FSSCALE, VTEM, IER)
DIMENSION U(*), USCALE(*), FVAL(*), FSSCALE(*), VTEM(*)
\end{verbatim}
Typically this routine will use only \( U, F\text{VAL} \), and \( V\text{TEM} \). It must solve the preconditioned linear system \( Pz = r \), where \( r = V\text{TEM} \) is input, and store the solution \( z \) in \( V\text{TEM} \) as well. Here \( P \) is the correct 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 \( US\text{CALE} \) and \( FS\text{CALE} \), 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:

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

It must perform any evaluation of Jacobian-related data and preprocessing needed for the solution of the preconditioned linear systems by \( FK\text{PSOL} \). The variables \( U \) through \( FS\text{SCALE} \) are for use in the preconditioning setup process. Typically, the system function \( FK\text{FUN} \) is called before any calls to \( FK\text{PSET} \), so that \( F\text{VAL} \) will have been updated. \( U \) is the current solution iterate. If scaling is being used, \( US\text{CALE} \) and \( FS\text{SCALE} \) are available for those operations requiring scaling.

On return, set \( IER = 0 \) if \( FK\text{PSET} \) was successful, or set \( IER = 1 \) if an error occurred.

To indicate that the \( FK\text{INPSET} \) and \( FK\text{INPSOL} \) routines are supplied, then the user must call

```fortran
CALL FKINLSSETPREC (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 \( FK\text{PSOL} \) and \( FK\text{PSET} \) (see below).

The previous routine \( FK\text{INPSILSETPREC} \) is now a wrapper for this routine, and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

If the user calls \( FK\text{INLSSETPREC} \), the routine \( FK\text{PSET} \) 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:

```fortran
CALL FKIN\text{SOL} (U, GLOBAL\text{STRAT}, USCALE, FSSCALE, IER)
```

The arguments are as follows. \( U \) is an array containing the initial guess on input, and the solution on return. \( GLOBAL\text{STRAT} \) is an integer (type \texttt{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). \( US\text{CALE} \) is an array of scaling factors for the \( U \) vector. \( FS\text{SCALE} \) is an array of scaling factors for the \( F\text{VAL} \) 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 \texttt{KIN}So1 returns (see \S 4.5.3 and \S B.2). The values of the optional outputs are available in \texttt{IOPT} and \texttt{ROPT} (see Table 5.2).

10. **Memory deallocation**

To free the internal memory created by calls to \( FK\text{INCREASE} \), \( FK\text{INIT} \), \( FN\text{ININIT*} \), \( FK\text{INLSINIT} \), and \( FS\text{UN***MATINIT} \), make the call

```fortran
CALL FKINFREE
```
5.5 FKINSOL optional input and output

Table 5.1: Keys for setting FKINSOL optional inputs

<table>
<thead>
<tr>
<th>Key</th>
<th>Optional input</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRNT_LEVEL</td>
<td>Verbosity level of output</td>
<td>0</td>
</tr>
<tr>
<td>MAA</td>
<td>Number of prior residuals for Anderson Acceleration</td>
<td>0</td>
</tr>
<tr>
<td>MAX_NITERS</td>
<td>Maximum no. of nonlinear iterations</td>
<td>200</td>
</tr>
<tr>
<td>ETA_FORM</td>
<td>Form of $\eta$ coefficient</td>
<td>1 ($\text{KIN_ETACHOICE1}$)</td>
</tr>
<tr>
<td>MAX_SETUPS</td>
<td>Maximum no. of iterations without prec. setup</td>
<td>10</td>
</tr>
<tr>
<td>MAX_SP_SETUPS</td>
<td>Maximum no. of iterations without residual check</td>
<td>5</td>
</tr>
<tr>
<td>NO_INIT_SETUP</td>
<td>No initial preconditioner setup</td>
<td>SUNFALSE</td>
</tr>
<tr>
<td>NO_MIN_EPS</td>
<td>Lower bound on $\epsilon$</td>
<td>SUNFALSE</td>
</tr>
<tr>
<td>NO_RES_MON</td>
<td>No residual monitoring</td>
<td>SUNFALSE</td>
</tr>
</tbody>
</table>

Real optional inputs (FKINSETRIN)

<table>
<thead>
<tr>
<th>Key</th>
<th>Optional input</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FNORM_TOL</td>
<td>Function-norm stopping tolerance</td>
<td>uround$^{1/3}$</td>
</tr>
<tr>
<td>SSTEP_TOL</td>
<td>Scaled-step stopping tolerance</td>
<td>uround$^{2/3}$</td>
</tr>
<tr>
<td>MAX_STEP</td>
<td>Max. scaled length of Newton step</td>
<td>$1000|D_u u_0|_2$</td>
</tr>
<tr>
<td>RERR_FUNC</td>
<td>Relative error for F.D. $J_v$</td>
<td>uround$^{1/3}$</td>
</tr>
<tr>
<td>ETA_CONST</td>
<td>Constant value of $\eta$</td>
<td>0.1</td>
</tr>
<tr>
<td>ETA_PARAMS</td>
<td>Values of $\gamma$ and $\alpha$</td>
<td>0.9 and 2.0</td>
</tr>
<tr>
<td>RMON_CONST</td>
<td>Constant value of $\omega$</td>
<td>0.9</td>
</tr>
<tr>
<td>RMON_PARAMS</td>
<td>Values of $\omega_{\text{min}}$ and $\omega_{\text{max}}$</td>
<td>0.00001 and 0.9</td>
</tr>
</tbody>
</table>

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:

```fortran
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 CONST_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 FKINBBBD interface to KINBBBDPRE

The FKINBBBD 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
Table 5.2: Description of the FKINSOL optional output arrays IOUT and ROUT

### Integer output array IOUT

<table>
<thead>
<tr>
<th>Index</th>
<th>Optional output</th>
<th>KINSOL function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LENRW</td>
<td>KINGetWorkSpace</td>
</tr>
<tr>
<td>2</td>
<td>LENIW</td>
<td>KINGetWorkSpace</td>
</tr>
<tr>
<td>3</td>
<td>NNI</td>
<td>KINGetNumNonlinSolvIters</td>
</tr>
<tr>
<td>4</td>
<td>NFE</td>
<td>KINGetNumFuncEvals</td>
</tr>
<tr>
<td>5</td>
<td>NBCF</td>
<td>KINGetNumBetaCondFails</td>
</tr>
<tr>
<td>6</td>
<td>NBKTRK</td>
<td>KINGetNumBacktrackOps</td>
</tr>
</tbody>
</table>

#### KINLS linear solver interface

<table>
<thead>
<tr>
<th>Index</th>
<th>Optional output</th>
<th>KINSOL function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>LENRWLS</td>
<td>KINGetLinWorkSpace</td>
</tr>
<tr>
<td>8</td>
<td>LENIWLS</td>
<td>KINGetLinWorkSpace</td>
</tr>
<tr>
<td>9</td>
<td>LS_FLAG</td>
<td>KINGetLastLinFlag</td>
</tr>
<tr>
<td>10</td>
<td>NFELS</td>
<td>KINGetNumLinFuncEvals</td>
</tr>
<tr>
<td>11</td>
<td>NJE</td>
<td>KINGetNumJacEvals</td>
</tr>
<tr>
<td>12</td>
<td>NJTV</td>
<td>KINGetNumJtimesEvals</td>
</tr>
<tr>
<td>13</td>
<td>NPE</td>
<td>KINGetNumPrecEvals</td>
</tr>
<tr>
<td>14</td>
<td>NPS</td>
<td>KINGetNumPrecSolves</td>
</tr>
<tr>
<td>15</td>
<td>NLI</td>
<td>KINGetNumLinIters</td>
</tr>
<tr>
<td>16</td>
<td>NCFL</td>
<td>KINGetNumLinConvFails</td>
</tr>
</tbody>
</table>

### Real output array ROUT

<table>
<thead>
<tr>
<th>Index</th>
<th>Optional output</th>
<th>KINSOL function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FNORM</td>
<td>KINGetFuncNorm</td>
</tr>
<tr>
<td>2</td>
<td>SSTEP</td>
<td>KINGetStepLength</td>
</tr>
</tbody>
</table>
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:

- **FKINBBBDINIT** 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):

<table>
<thead>
<tr>
<th>Fkinbbd routine (FORTRAN, user-supplied)</th>
<th>KINSOL function (C, interface)</th>
<th>KINSOL type of interface function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FKLOCFN</td>
<td>FKINGloc</td>
<td>KINBBDLocalFn</td>
</tr>
<tr>
<td>FKCMMF</td>
<td>FKINGcomm</td>
<td>KINBBDCommFn</td>
</tr>
<tr>
<td>FKJTIMES</td>
<td>FkinJtimes</td>
<td>KINLSJacTimesVecFn</td>
</tr>
</tbody>
</table>

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. Solver Initialization
7. **Linear solver interface specification**
   - Initialize the **KINLS** iterative linear solver interface by calling FKINLSINIT.

To initialize the KINBBDPRE preconditioner, make the following call:

```fortran
CALL FkinBBBDINIT (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 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 FKINLSSETJAC (FLAG, IER)

with $\text{FLAG} \neq 0$. (See step 8 in §5.4).

8. Problem solution

9. 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, NGEBBBD)

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. NGEBBBD is the cumulative number of $G(u)$ evaluations (calls to FKLOCFN) so far.

10. Memory deallocation

(The memory allocated for the FKINBBD module is deallocated automatically by FKINFREE.)

11. User-supplied routines

The following two routines must be supplied for use with the KINBBDPRE module:

\[
\text{SUBROUTINE FKLOCFN (NLOC, ULOC, GLOC, IER)}
\]
\[
\text{DIMENSION ULOC(*), GLOC(*)}
\]

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).

\[
\text{SUBROUTINE FKCOMMFN (NLOC, ULOC, IER)}
\]
\[
\text{DIMENSION ULOC(*)}
\]

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 NVECTORModule

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 (*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);
};
Description of the NVECTOR module

The generic NVECTOR module defines and implements the vector operations acting on an 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:

```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 standard vector operations defined by the generic NVECTOR module. Tables 6.3 and 6.4 list optional fused and vector array operations respectively. These operations are intended to increase data reuse, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. If a particular NVECTOR implementation defines one of the fused or vector array operations as NULL, the NVECTOR interface will call one of the standard vector operations as necessary.

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

```c
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

```c
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:
Table 6.1: Vector Identifications associated with vector kernels supplied with SUNDIALS.

<table>
<thead>
<tr>
<th>Vector ID</th>
<th>Vector type</th>
<th>ID Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNDIALS_NVEC_SERIAL</td>
<td>Serial</td>
<td>0</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_PARALLEL</td>
<td>Distributed memory parallel (MPI)</td>
<td>1</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_OPENMP</td>
<td>OpenMP shared memory parallel</td>
<td>2</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_PTHREADS</td>
<td>PThreads shared memory parallel</td>
<td>3</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_PARHYP</td>
<td>hypre ParHyp parallel vector</td>
<td>4</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_PETSC</td>
<td>PETSc parallel vector</td>
<td>5</td>
</tr>
<tr>
<td>SUNDIALS_NVEC_CUSTOM</td>
<td>User-provided custom vector</td>
<td>6</td>
</tr>
</tbody>
</table>

- 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 NVVECTOR 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 NVVECTOR implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 6.1. It is recommended that a user-supplied NVVECTOR implementation use the SUNDIALS_NVEC_CUSTOM identifier.
Table 6.2: Description of the NVector operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVGetVectorID</td>
<td>id = NVGetVectorID(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 NV vector interface. Returned values are given in Table 6.1.</td>
</tr>
<tr>
<td>NVClone</td>
<td>v = NVClone(w); Creates a new NV 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.</td>
</tr>
<tr>
<td>NVCloneEmpty</td>
<td>v = NVCloneEmpty(w); Creates a new NV vector of the same type as an existing vector w and sets the ops field. It does not allocate storage for data.</td>
</tr>
<tr>
<td>NVDestroy</td>
<td>NVDestroy(v); Destroys the NV vector v and frees memory allocated for its internal data.</td>
</tr>
<tr>
<td>NVSpace</td>
<td>NVSpace(nvSpec, &amp;lrw, &amp;liw); Returns storage requirements for one NV 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 NV vector module if that information is not of interest.</td>
</tr>
<tr>
<td>NVGetArrayPointer</td>
<td>vdata = NVGetArrayPointer(v); Returns a pointer to a realtype array from the NV vector v. Note that this assumes that the internal data in NV 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.</td>
</tr>
<tr>
<td>NVSetArrayPointer</td>
<td>NVSetArrayPointer(vdata, v); Overwrites the data in an NV vector with a given array of realtype. Note that this assumes that the internal data in NV 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 NV vector module for a parallel environment.</td>
</tr>
</tbody>
</table>

continued on next page
<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_VLinearSum</td>
<td>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 ).</td>
</tr>
<tr>
<td>N_VConst</td>
<td>N_VConst(c, z); Sets all components of the N_Vector ( z ) to realtype ( c ): ( z_i = c, i = 0, \ldots, n - 1 ).</td>
</tr>
<tr>
<td>N_VProd</td>
<td>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 ).</td>
</tr>
<tr>
<td>N_VDiv</td>
<td>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.</td>
</tr>
<tr>
<td>N_VScale</td>
<td>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 ).</td>
</tr>
<tr>
<td>N_VAbs</td>
<td>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 =</td>
</tr>
<tr>
<td>N_VInv</td>
<td>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.</td>
</tr>
<tr>
<td>N_VAddConst</td>
<td>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 ).</td>
</tr>
<tr>
<td>N_VDotProd</td>
<td>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 ).</td>
</tr>
<tr>
<td>N_VMaxNorm</td>
<td>m = N_VMaxNorm(x); Returns the maximum norm of the N_Vector ( x ): ( m = \max_i</td>
</tr>
</tbody>
</table>
## Description of the NVECTOR module

### Name | Usage and Description
---|---
N_VWrmsNorm | \( m = \text{N}_V\text{WrmsNorm}(x, w) \)

Returns the weighted root-mean-square norm of the \( \text{N}\_\text{Vector} \) \( x \) with \texttt{realtype} weight vector \( w \): \[ m = \sqrt{\frac{\sum_{i=0}^{n-1} (x_i w_i)^2}{n}}. \]

N_VWrmsNormMask | \( m = \text{N}_V\text{WrmsNormMask}(x, w, \text{id}) \)

Returns the weighted root mean square norm of the \( \text{N}\_\text{Vector} \) \( x \) with \texttt{realtype} weight vector \( w \) built using only the elements of \( x \) corresponding to positive elements of the \( \text{N}\_\text{Vector} \) \( \text{id} \):

\[
m = \sqrt{\frac{\sum_{i=0}^{n-1} (x_i w_i H(id_i))^2}{n}}, \quad \text{where } H(\alpha) = \begin{cases} 1 & \alpha > 0 \\ 0 & \alpha \leq 0 \end{cases}
\]

N_VMin | \( m = \text{N}_V\text{Min}(x) \)

Returns the smallest element of the \( \text{N}\_\text{Vector} \) \( x \):

\[ m = \min_i x_i. \]

N_VL2Norm | \( m = \text{N}_V\text{L2Norm}(x, w) \)

Returns the weighted Euclidean \( \ell_2 \) norm of the \( \text{N}\_\text{Vector} \) \( x \) with \texttt{realtype} weight vector \( w \):

\[ m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2}. \]

N_VL1Norm | \( m = \text{N}_V\text{L1Norm}(x) \)

Returns the \( \ell_1 \) norm of the \( \text{N}\_\text{Vector} \) \( x \):

\[ m = \sum_{i=0}^{n-1} |x_i|. \]

N_VCompare | \( \text{N}_V\text{Compare}(c, x, z) \)

Compares the components of the \( \text{N}\_\text{Vector} \) \( x \) to the \texttt{realtype} scalar \( c \) and returns an \( \text{N}\_\text{Vector} \) \( z \) such that: \[ z_i = 1.0 \text{ if } |x_i| \geq c \text{ and } z_i = 0.0 \text{ otherwise.} \]

N_VInvTest | \( t = \text{N}_V\text{InvTest}(x, z) \)

Sets the components of the \( \text{N}\_\text{Vector} \) \( z \) to be the inverses of the components of the \( \text{N}\_\text{Vector} \) \( x \), with prior testing for zero values: \( z_i = 1.0/x_i, i = 0, \ldots, n - 1. \) This routine returns a boolean assigned to \texttt{SUNTRUE} if all components of \( x \) are nonzero (successful inversion) and returns \texttt{SUNFALSE} otherwise.

N_VConstrMask | \( t = \text{N}_V\text{ConstrMask}(c, x, m) \)

Performs the following constraint tests: \( x_i > 0 \text{ if } c_i = 2, \ x_i \geq 0 \text{ if } c_i = 1, \ x_i \leq 0 \text{ if } c_i = -1, \ x_i < 0 \text{ if } c_i = -2. \) There is no constraint on \( x_i \) if \( c_i = 0. \) This routine returns a boolean assigned to \texttt{SUNFALSE} if any element failed the constraint test and assigned to \texttt{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.

---

*continued on next page*
Name: **N_VMinQuotient**

```c
minq = N_VMinQuotient(num, denom);
```

This routine returns the minimum of the quotients obtained by term-wise dividing `num` by `denom`. 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.

### Table 6.3: Description of the NVECTOR fused operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
</table>
| **N_VLinearCombination** | `ier = N_VLinearCombination(nv, c, X, z);`<br>This routine computes the linear combination of `nv` vectors with `n` elements:<br>
```c
z_i = \sum_{j=0}^{n_v-1} c_j x_{j,i}, \quad i = 0, \ldots, n - 1,
```
where `c` is an array of `n_v` scalars (type `realtype`), `X` is an array of `n_v` vectors (type `N_Vector`), and `z` is the output vector (type `N_Vector`). If the output vector `z` is one of the vectors in `X`, then it must be the first vector in the vector array. The operation returns 0 for success and a non-zero value otherwise. |
| **N_VScaleAddMulti**     | `ier = N_VScaleAddMulti(nv, c, x, Y, Z);`<br>This routine scales and adds one vector to `nv` vectors with `n` elements:<br>
```c
z_{j,i} = c_j x_i + y_{j,i}, \quad j = 0, \ldots, n_v - 1 \quad i = 0, \ldots, n - 1,
```
where `c` is an array of `n_v` scalars (type `realtype`), `x` is the vector (type `N_Vector`) to be scaled and added to each vector in the vector array of `n_v` vectors `Y` (type `N_Vector`), and `Z` (type `N_Vector`) is a vector array of `n_v` output vectors. The operation returns 0 for success and a non-zero value otherwise. |
### Description of the NVECTOR module

**N_VDotProdMulti**

```c
ing = N_VDotProdMulti(nv, x, Y, d);
```

This routine computes the dot product of a vector with `nv` other vectors:

\[
d_j = \sum_{i=0}^{n-1} x_i y_{j,i}, \quad j = 0, \ldots, nv - 1,
\]

where `d` (type `realtype*`) is an array of `nv` scalars containing the dot products of the vector `x` (type `N_Vector`) with each of the `nv` vectors in the vector array `Y` (type `N_Vector*`). The operation returns 0 for success and a non-zero value otherwise.

### Table 6.4: Description of the NVECTOR vector array operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>N_VLinearSumVectorArray</strong></td>
<td><code>ier = N_VLinearSumVectorArray(nv, a, X, b, Y, Z);</code></td>
</tr>
<tr>
<td></td>
<td>This routine computes the linear sum of two vector arrays containing <code>nv</code> vectors of <code>n</code> elements:</td>
</tr>
</tbody>
</table>
|                       | \[
|                       | \quad z_{j,i} = ax_{j,i} + by_{j,i}, \quad i = 0, \ldots, n - 1 \quad j = 0, \ldots, nv - 1, |
|                       | where `a` and `b` are `realtype` scalars and `X`, `Y`, and `Z` are arrays of `nv` vectors (type `N_Vector*`). The operation returns 0 for success and a non-zero value otherwise. |
| **N_VScaleVectorArray** | `ier = N_VScaleVectorArray(nv, c, X, Z);`                                            |
|                       | This routine scales each vector of `n` elements in a vector array of `nv` vectors by a potentially different constant: |
|                       | \[
|                       | \quad z_{j,i} = cx_{j,i}, \quad i = 0, \ldots, n - 1 \quad j = 0, \ldots, nv - 1, |
|                       | where `c` is an array of `nv` scalars (type `realtype*`) and `X` and `Z` are arrays of `nv` vectors (type `N_Vector*`). The operation returns 0 for success and a non-zero value otherwise. |

*continued on next page*
<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
</table>
| **N_VConstVectorArray** | `ier = N_VConstVectorArray(nv, c, X);`  
This routine sets each element in a vector of `n` elements in a vector array of `nv` vectors to the same value:  
\[ z_{j,i} = c, \quad i = 0,\ldots, n-1 \quad j = 0,\ldots, nv - 1, \]  
where `c` is a `realtype` scalar and `X` is an array of `nv` vectors (type `N_Vector`). The operation returns 0 for success and a non-zero value otherwise. |
| **N_VWrmsNormVectorArray** | `ier = N_VWrmsNormVectorArray(nv, X, W, m);`  
This routine computes the weighted root mean square norm of `nv` vectors with `n` elements:  
\[ m_j = \left( \frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i}w_{j,i})^2 \right)^{1/2}, \quad j = 0,\ldots, nv - 1, \]  
where `m` (type `realtype`) contains the `nv` norms of the vectors in the vector array `X` (type `N_Vector`) with corresponding weight vectors `W` (type `N_Vector`). The operation returns 0 for success and a non-zero value otherwise. |
| **N_VWrmsNormMaskVectorArray** | `ier = N_VWrmsNormMaskVectorArray(nv, X, W, id, m);`  
This routine computes the masked weighted root mean square norm of `nv` vectors with `n` elements:  
\[ m_j = \left( \frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i}w_{j,i}H(id_i))^2 \right)^{1/2}, \quad j = 0,\ldots, nv - 1, \]  
\[ H(id_i) = 1 \text{ for } id_i > 0 \text{ and is zero otherwise}, \]  
`m` (type `realtype`) contains the `nv` norms of the vectors in the vector array `X` (type `N_Vector`) with corresponding weight vectors `W` (type `N_Vector`) and mask vector `id` (type `N_Vector`). The operation returns 0 for success and a non-zero value otherwise. |
### Description of the NVECTOR module

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
</table>
| N_VScaleAddMultiVectorArray | \( \text{ier} = \text{N_VScaleAddMultiVectorArray}(\text{nv}, \text{ns}, c, X, YY, ZZ); \)  
This routine scales and adds a vector in a vector array of \( n_v \) vectors to the corresponding vector in \( n_s \) vector arrays:  
\[
z_{j,i} = \sum_{k=0}^{n_s-1} c_k x_{k,j,i}, \quad i = 0, \ldots, n-1 \quad j = 0, \ldots, n_v - 1,
\] 
where \( c \) is an array of \( n_s \) scalars (type \text{realtype*}), \( X \) is a vector array of \( n_v \) vectors (type \text{idN_Vector*}) to be scaled and added to the corresponding vector in each of the \( n_s \) vector arrays in the array of vector arrays \( YY \) (type \text{N_Vector**}) and stored in the output array of vector arrays \( ZZ \) (type \text{N_Vector**}). The operation returns 0 for success and a non-zero value otherwise. |
| N_VLinearCombinationVectorArray | \( \text{ier} = \text{N_VLinearCombinationVectorArray}(\text{nv}, \text{ns}, c, XX, Z); \)  
This routine computes the linear combination of \( n_s \) vector arrays containing \( n_v \) vectors with \( n \) elements:  
\[
z_{j,i} = \sum_{k=0}^{n_s-1} c_k x_{k,j,i}, \quad i = 0, \ldots, n-1 \quad j = 0, \ldots, n_v - 1,
\] 
where \( c \) is an array of \( n_s \) scalars (type \text{realtype*}), \( XX \) (type \text{N_Vector**}) is an array of \( n_s \) vector arrays each containing \( n_v \) vectors to be summed into the output vector array of \( n_v \) vectors \( Z \) (type \text{N_Vector*}). If the output vector array \( Z \) is one of the vector arrays in \( XX \), then it must be the first vector array in \( XX \). The operation returns 0 for success and a non-zero value otherwise. |

#### 6.1 The NVECTOR_SERIAL implementation

The serial implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_SERIAL, defines the `content` field of \text{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`.

```c
struct _N_VectorContent_Serial {
    sunindextype length;
    boolean  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`
6.1 The NVECTOR_SERIAL implementation

This routine gives access to the contents of the serial vector N_Vector.

The assignment \( v_{\text{cont}} = \text{NVCONTENT}_S(v) \) sets \( v_{\text{cont}} \) to be a pointer to the serial N_Vector content structure.

Implementation:

```c
#define NV_CONTENT_S(v) ( (N_VectorContent_Serial)(v->content) )
```

- \( \text{NVOWNDATA}_S, \text{NVDATA}_S, \text{NVLENGTH}_S \)

These macros give individual access to the parts of the content of a serial N_Vector.

The assignment \( v_{\text{data}} = \text{NVDATA}_S(v) \) sets \( v_{\text{data}} \) to be a pointer to the first component of the data for the N_Vector \( v \). The assignment \( \text{NVDATA}_S(v) = v_{\text{data}} \) sets the component array of \( v \) to be \( v_{\text{data}} \) by storing the pointer \( v_{\text{data}} \).

The assignment \( v_{\text{len}} = \text{NVLENGTH}_S(v) \) sets \( v_{\text{len}} \) to be the length of \( v \). On the other hand, the call \( \text{NVLENGTH}_S(v) = \text{len}_v \) sets the length of \( v \) to be \( \text{len}_v \).

Implementation:

```c
#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 )
```

- \( \text{NVIth}_S \)

This macro gives access to the individual components of the data array of an N_Vector.

The assignment \( r = \text{NVIth}_S(v,i) \) sets \( r \) to be the value of the \( i \)-th component of \( v \). The assignment \( \text{NVIth}_S(v,i) = r \) 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:

```c
#define NV_Ith_S(v,i) ( NV_DATA_S(v)[i] )
```

The nvector_serial module defines serial implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4. Their names are obtained from those in Tables 6.2, 6.3, and 6.4 by appending the suffix _Serial (e.g. NVDestroy_Serial). The module NVECTOR_SERIAL provides the following additional user-callable routines:

- \( \text{NNEW}_\text{Serial} \)

This function creates and allocates memory for a serial N_Vector. Its only argument is the vector length.

```c
N_Vector N_NVNew_Serial(sunindextype vec_length);
```

- \( \text{NVNEWEMPTY}_\text{Serial} \)

This function creates a new serial N_Vector with an empty (NULL) data array.

```c
N_Vector N_NVNewEmpty_Serial(sunindextype vec_length);
```

- \( \text{NMAKE}_\text{Serial} \)

This function creates and allocates memory for a serial vector with user-provided data array.

(This function does not allocate memory for \( v_{\text{data}} \) itself.)

```c
N_Vector N_NMake_Serial(sunindextype vec_length, realtype *v_data);
```

- \( \text{NVCLONEVECTORARRAY}_\text{Serial} \)

This function creates (by cloning) an array of count serial vectors.

```c
N_Vector *N_NVCloneVectorArray_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.
  
  ```c
  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`.
  
  ```c
  void N_VDestroyVectorArray_Serial(N_Vector *vs, int count);
  ```

• **N_VGetLength_Serial**
  
  This function returns the number of vector elements.
  
  ```c
  sunindextype N_VGetLength_Serial(N_Vector v);
  ```

• **N_VPrint_Serial**
  
  This function prints the content of a serial vector to `stdout`.
  
  ```c
  void N_VPrint_Serial(N_Vector v);
  ```

• **N_VPrintFile_Serial**
  
  This function prints the content of a serial vector to `outfile`.
  
  ```c
  void N_VPrintFile_Serial(N_Vector v, FILE *outfile);
  ```

**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 `NV VECTOR_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 `NV VECTOR_SERIAL` module also includes a Fortran-callable function `FNVINITS(code, NEQ, IER)`, to initialize this `NV VECTOR_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 `NV VECTOR_PARALLEL` implementation

The `NV VECTOR_PARALLEL` implementation of the `NV VECTOR` 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`. 
6.2 The NVECTOR_PARALLEL implementation

```c
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:
  ```c
  #define NV_CONTENT_P(v) ( (N_VectorContent_Parallel)(v->content) )
  ```

  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 `vllen = NV_LOCLENGTH_P(v)` sets `vllen` 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 `vglen = NV_GLOBLENGTH_P(v)` sets `vglen` 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:
  ```c
  #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:
  ```c
  #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:
  ```c
  #define NV_Ith_P(v,i) ( NV_DATA_P(v)[i] )
  ```
The nvector_parallel module defines parallel implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4. Their names are obtained from those in Tables 6.2, 6.3, and 6.4 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.

  \[
  \text{N-Vector N_VNew_Parallel(MPI_Comm \ comm,} \\
  \quad \quad \text{sunindextype local_length,} \\
  \quad \quad \text{sunindextype global_length);}
  \]

- **N_VNewEmpty_Parallel**
  This function creates a new parallel N_Vector with an empty (NULL) data array.

  \[
  \text{N-Vector N_VNewEmpty_Parallel(MPI_Comm \ comm,} \\
  \quad \quad \text{sunindextype local_length,} \\
  \quad \quad \text{sunindextype global_length);}
  \]

- **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 \text{v.data} itself.)

  \[
  \text{N-Vector N_VMake_Parallel(MPI_Comm \ comm,} \\
  \quad \quad \text{sunindextype local_length,} \\
  \quad \quad \text{sunindextype global_length,} \\
  \quad \quad \text{realtype *v_data);}
  \]

- **N_VCloneVectorArray_Parallel**
  This function creates (by cloning) an array of count parallel vectors.

  \[
  \text{N-Vector *N_VCloneVectorArray_Parallel(int count, N-Vector w);} \\
  \text{)}
  \]

- **N_VCloneVectorArrayEmpty_Parallel**
  This function creates (by cloning) an array of count parallel vectors, each with an empty (NULL) data array.

  \[
  \text{N-Vector *N_VCloneVectorArrayEmpty_Parallel(int count, N-Vector w);} \\
  \text{)}
  \]

- **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);
6.3 The NVECTOR_OPENMP implementation

- **N_VPrint_Parallel**
  This function prints the local content of a parallel vector to stdout.
  ```c
  void N_VPrint_Parallel(N_Vector v);
  ```

- **N_VPrintFile_Parallel**
  This function prints the local content of a parallel vector to outfile.
  ```c
  void N_VPrintFile_Parallel(N_Vector v, FILE *outfile);
  ```

**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.

```c
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_nvecomopenmp.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_OM**
  
  This routine gives access to the contents of the OpenMP vector `N_Vector`. The assignment `v_cont = NV_CONTENT_OM(v)` sets `v_cont` to be a pointer to the OpenMP `N_Vector` content structure.
  
  Implementation:
  ```c
  #define NV_CONTENT_OM(v) ( (N_VectorContent_OpenMP)(v->content) )
  ```

- **NV_OWN_DATA_OM**, **NV_DATA_OM**, **NV_LENGTH_OM**, **NV_NUM_THREADS_OM**
  
  These macros give individual access to the parts of the content of a OpenMP `N_Vector`.
  
  The assignment `v_data = NV_DATA_OM(v)` sets `v_data` to be a pointer to the first component of the data for the `N_Vector` `v`. The assignment `NV_DATA_OM(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_OM(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_OM(v) = len` sets the length of `v` to be `len`.
  
  The assignment `v_num_threads = NV_NUM_THREADS_OM(v)` sets `v_num_threads` to be the number of threads from `v`. On the other hand, the call `NV_NUM_THREADS_OM(v) = num_threads` sets the number of threads for `v` to be `num_threads`.
  
  Implementation:
  ```c
  #define NV_OWN_DATA_OM(v) ( NV_CONTENT_OM(v)->own_data )
  #define NV_DATA_OM(v) ( NV_CONTENT_OM(v)->data )
  #define NV_LENGTH_OM(v) ( NV_CONTENT_OM(v)->length )
  #define NV_NUM_THREADS_OM(v) ( NV_CONTENT_OM(v)->num_threads )
  ```

- **NV_Ith_OM**
  
  This macro gives access to the individual components of the data array of an `N_Vector`.
  
  The assignment `r = NV_Ith_OM(v, i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_OM(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:
  ```c
  #define NV_Ith_OM(v, i) ( NV_DATA_OM(v)[i] )
  ```

The `nvector_openmp` module defines OpenMP implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4. Their names are obtained from those in Tables 6.2, 6.3, and 6.4 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.
  ```c
  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.
  ```c
  N_Vector N_VNewEmpty_OpenMP(sunindextype vec_length, int num_threads);
  ```
6.3 The NVECTOR_OPENMP implementation

- **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 \texttt{v.data} itself.)

  \[
  \text{N.Vector N_VMake_OpenMP(sunindextype vec_length, realtype *v.data, int num_threads);} \]

- **N_VCloneVectorArray_OpenMP**
  This function creates (by cloning) an array of count OpenMP vectors.

  \[
  \text{N.Vector *}N_{\text{VCloneVectorArray}}_{\text{OpenMP}}(\text{int count, N.Vector w}) ; \]

- **N_VCloneVectorArrayEmpty_OpenMP**
  This function creates (by cloning) an array of count OpenMP vectors, each with an empty (NULL) data array.

  \[
  \text{N.Vector *}N_{\text{VCloneVectorArrayEmpty}}_{\text{OpenMP}}(\text{int count, N.Vector w}) ; \]

- **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.

  \[
  \text{void N_VDestroyVectorArray} _{\text{OpenMP}}(\text{N.Vector *vs, int count}) ; \]

- **N_VGetLength_OpenMP**
  This function returns number of vector elements.

  \[
  \text{sunindextype N_VGetLength} _{\text{OpenMP}}(\text{N.Vector v}) ; \]

- **N_VPrint_OpenMP**
  This function prints the content of an OpenMP vector to stdout.

  \[
  \text{void N_VPrint} _{\text{OpenMP}}(\text{N.Vector v}) ; \]

- **N_VPrintFile_OpenMP**
  This function prints the content of an OpenMP vector to outfile.

  \[
  \text{void N_VPrintFile} _{\text{OpenMP}}(\text{N.Vector v, FILE *outfile}) ; \]

**Notes**

- When looping over the components of an N.Vector \textit{v}, it is more efficient to first obtain the component array via \texttt{v.data = NV_DATA_OMP(v)} and then access \texttt{v.data[i]} within the loop than it is to use \texttt{NV_Ith_OMP(v,i)} within the loop.

- N_VNewEmpty_OpenMP, N_VMake_OpenMP, and N_VCloneVectorArrayEmpty_OpenMP set the field \texttt{own.data = SUNFALSE}. N_VDestroy_OpenMP and N_VDestroyVectorArray_OpenMP will not attempt to free the pointer \texttt{data} for any N.Vector with \texttt{own.data} set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the \texttt{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 \texttt{FNVINITOMP(code, NEQ, NUMTHREADS, IER)}, to initialize this module. Here \texttt{code} is an input solver id (1 for \texttt{cvode}, 2 for \texttt{ida}, 3 for \texttt{kinsol}, 4 for \texttt{arkode}); \texttt{NEQ} is the problem size (declared so as to match C type \texttt{long int}); \texttt{NUMTHREADS} is the number of threads; and \texttt{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 \( v_{\text{cont}} = \text{NV\_CONTENT\_PT}(v) \) sets \( v_{\text{cont}} \) 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 \( v_{\text{data}} = \text{NV\_DATA\_PT}(v) \) sets \( v_{\text{data}} \) to be a pointer to the first component of the data for the N_Vector \( v \). The assignment \( \text{NV\_DATA\_PT}(v) = v_{\text{data}} \) sets the component array of \( v \) to be \( v_{\text{data}} \) by storing the pointer \( v_{\text{data}} \).

  The assignment \( v_{\text{len}} = \text{NV\_LENGTH\_PT}(v) \) sets \( v_{\text{len}} \) to be the length of \( v \). On the other hand, the call \( \text{NV\_LENGTH\_PT}(v) = \text{len} \) sets the length of \( v \) to be \( \text{len} \).

  The assignment \( v_{\text{num\_threads}} = \text{NV\_NUM\_THREADS\_PT}(v) \) sets \( v_{\text{num\_threads}} \) to be the number of threads from \( v \). On the other hand, the call \( \text{NV\_NUM\_THREADS\_PT}(v) = \text{num\_threads} \) sets the number of threads for \( v \) to be \( \text{num\_threads} \).

  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 $r = \text{NV}_i^\text{th}_\text{PT}(v,i)$ sets $r$ to be the value of the $i$-th component of $v$. The assignment $\text{NV}_i^\text{th}_\text{PT}(v,i) = r$ 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:

```c
#define NV_Ith_PT(v,i) ( NV_DATA_PT(v)[i] )
```

The `nvector pthreads` module defines Pthreads implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4. Their names are obtained from those in Tables 6.2, 6.3, and 6.4 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.
  ```c
  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.
  ```c
  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.)
  ```c
  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.
  ```c
  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.
  ```c
  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`.
  ```c
  void N_VDestroyVectorArray_Pthreads(N_Vector *vs, int count);
  ```

- **N_VGetLength_Pthreads**
  This function returns the number of vector elements.
  ```c
  sunindextype N_VGetLength_Pthreads(N_Vector v);
  ```

- **N_VPrint_Pthreads**
  This function prints the content of a Pthreads vector to `stdout`.
  ```c
  void N_VPrint_Pthreads(N_Vector v);
  ```

- **N_VPrintFile_Pthreads**
  This function prints the content of a Pthreads vector to `outfile`.
  ```c
  void N_VPrintFile_Pthreads(N_Vector v, FILE *outfile);
  ```
Description of the NVECTOR module

Notes

- When looping over the components of an N_Vector v, it is more efficient to first obtain the component array via \( v\text{\textunderscore data} = \text{NV\textunderscore DATA\textunderscore PT}(v) \) and then access \( v\text{\textunderscore data}[i] \) within the loop than it is to use \( \text{NV\textunderscore Ith\textunderscore PT}(v,i) \) within the loop.

- \( \text{N\textunderscore VNewEmpty\textunderscore Pthreads}, \text{N\textunderscore VMake\textunderscore Pthreads}, \text{and N\textunderscore VCloneVectorArrayEmpty\textunderscore Pthreads} \) set the field \( \text{own\textunderscore data} = \text{SUNFALSE} \). \( \text{N\textunderscore VDestroy\textunderscore Pthreads} \) and \( \text{N\textunderscore VDestroyVectorArray\textunderscore Pthreads} \) will not attempt to free the pointer \( \text{data} \) for any N_Vector with \( \text{own\textunderscore data} \) set to SUNFALSE. In such a case, it is the user’s responsibility to deallocate the \( \text{data} \) 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 \( \text{FNVINITPTS}(\text{code, NEQ, NUMTHREADS, IER}) \), to initialize this module. Here \( \text{code} \) 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 \text{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 lib sundials_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 Tables 6.2, 6.3, and 6.4, except for \( \text{N\textunderscore VSetArrayPointer} \) and \( \text{N\textunderscore VGetArrayPointer} \), 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 Tables 6.2, 6.3, and 6.4 by appending the suffix \_ParHyp (e.g. \( \text{N\textunderscore VDestroy\textunderscore ParHyp} \)). The module NVECTOR\_PARHYP provides the following additional user-callable routines:

- \( \text{N\textunderscore VNewEmpty\textunderscore ParHyp} \)

  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 local content of a parhyp vector to stdout.

  void N_VPrint_ParHyp(N_Vector v);

- **N_VPrintFile_ParHyp**
  This function prints the local content of a parhyp vector to outfile.

  void N_VPrintFile_ParHyp(N_Vector v, FILE *outfile);

**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_Destroy_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.

```c
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 Tables 6.2, 6.3, and 6.4, 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 Tables 6.2, 6.3, and 6.4 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.

  ```c
  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.

  ```c
  N_Vector N_VMake_Petsc(Vec *pvec);
  ```

- **N_VGetVector_Petsc**
  This function returns a pointer to the underlying PETSc vector.

  ```c
  Vec *N_VGetVector_Petsc(N_Vector v);
  ```

- **N_VCloneVectorArray_Petsc**
  This function creates (by cloning) an array of count NVECTOR_PETSC vectors.

  ```c
  N_Vector *N_VCloneVectorArray_Petsc(int count, N_Vector w);
  ```
6.7 The NVVECTOR_CUDA implementation

- **N_VCloneVectorArrayEmpty_Petsc**
  This function creates (by cloning) an array of count NVVECTOR_PETSC vectors, each with pointers to PETSC vectors set to (NULL).

  
  N_Vector *N_VCloneVectorArrayEmpty_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 global content of a wrapped PETSC vector to stdout.

  
  void N_VPrint_Petsc(N_Vector v);

- **N_VPrintFile_Petsc**
  This function prints the global content of a wrapped PETSC vector to fname.

  
  void N_VPrintFile_Petsc(N_Vector v, const char fname[]);

Notes

- When there is a need to access components of an N_Vector_Petsc, v, it is recommended 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 NVVECTOR_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 NVVECTOR_CUDA implementation

The NVVECTOR_CUDA module is an experimental NVVECTOR 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_;  
  ThreadPartitioning<T, I>* partStream_;  
  ThreadPartitioning<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 ThreadPartitioning implementations that handle thread partitioning for streaming and reduction vector kernels, and a boolean flag that signals if the vector owns the thread partitioning. The class Vector inherits from the empty structure

```c
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 nvector_cuda module can be utilized for single-node parallelism or in a distributed context with MPI. The header file to include when using this module for single-node parallelism is nvector_cuda.h. The header file to include when using this module in the distributed case is nvector_mpicuda.h. Note that only the nvector_cuda constructor signature differs between the two header files. The installed module libraries to link to are lib sundials nvecCUDA.lib in the single-node case, or lib sundials nvecmpicuda.lib in the distributed case. Only one of these libraries may be linked to when creating an executable or library. SUNDIALS must be built with MPI support if the distributed library is desired. The extension, .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. Instead, user should use the accessor functions in the namespace suncudavec.

- `getDevData(N_Vector v)`
  This function takes an N_Vector as an argument and returns a raw pointer to the vector data on the device (GPU). It is the user’s responsibility to ensure that the vector argument is of the correct N_Vector type.

- `getHostData(N_Vector v)`
  This function takes a N_Vector as an argument and returns a raw pointer to the vector data on the host (CPU memory). It is the user’s responsibility to ensure that the vector argument is of the correct N_Vector type.

- `getSize(N_Vector v)`
  Returns the vector’s local length.

- `getGlobalSize(N_Vector v)`
  Returns the vector’s global length.

- `getMPIComm(N_Vector v)`
  Takes a N_Vector as an argument and returns a sundials communicator of type

The nvector_cuda module defines implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4, except for N_VGetArrayPointer and N_VSetArrayPointer. As such, this vector cannot be used with the SUNDIALS Fortran interfaces, nor with the SUNDIALS direct solvers and preconditioners. Instead, 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 Tables 6.2, 6.3, and 6.4 by appending the suffix _Cuda (e.g. N_VDestroy_Cuda). The module nvector_cuda provides the following additional user-callable routines:
6.7 The NVECTOR_CUDA implementation

• N_VNew_Cuda

  Note: this function signature is defined in the header nvector_mpicuda.h and should be used when using this module in a distributed context. This function creates and allocates memory for a CUDA N_Vector. The memory is allocated on both host and device. Its arguments are local and global vector lengths, as well as the MPI communicator. Use this constructor with the libsundials_nvecmpicuda.lib library.

  \[
  \text{N\_Vector } N\_VNew\_Cuda(\text{MPI\_Comm comm,} \\
  \text{sunindextype local\_length,} \\
  \text{sunindextype global\_length);}
  \]

• N_VNew_Cuda

  Note: this function signature is defined in the header nvector_cuda.h and should be used when using this module for single-node parallelism. This function creates and allocates memory for a CUDA N_Vector on a single node. The memory is allocated on both host and device. Its only argument is vector length. Use this constructor with the libsundials_nveccuda.lib library.

  \[
  \text{N\_Vector } N\_VNew\_Cuda(\text{sunindextype length);}
  \]

• 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.

  \[
  \text{N\_Vector } N\_VNewEmpty\_Cuda(\text{sunindextype vec\_length);}
  \]

• 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.

  \[
  \text{N\_Vector } N\_VMake\_Cuda(N\_VectorContent\_Cuda c);
  \]

• N_VGetLength_Cuda

  This function returns the length of the vector.

  \[
  \text{sunindextype } N\_VGetLength\_Cuda(\text{N\_Vector } v);
  \]

• N_VGetHostArrayPointer_Cuda

  This function returns a pointer to the vector data on the host.

  \[
  \text{realtype } *N\_VGetHostArrayPointer\_Cuda(\text{N\_Vector } v);
  \]

• N_VGetDeviceArrayPointer_Cuda

  This function returns a pointer to the vector data on the device.

  \[
  \text{realtype } *N\_VGetDeviceArrayPointer\_Cuda(\text{N\_Vector } v);
  \]

• N_VCopyToDevice_Cuda

  This function copies host vector data to the device.

  \[
  \text{realtype } *N\_VCopyToDevice\_Cuda(\text{N\_Vector } v);
  \]

• N_VCopyFromDevice_Cuda

  This function copies vector data from the device to the host.

  \[
  \text{realtype } *N\_VCopyFromDevice\_Cuda(\text{N\_Vector } v);
  \]
• N_VPrint_Cuda
  This function prints the content of a CUDA vector to stdout.
  void N_VPrint_Cuda(N_Vector v);

• N_VPrintFile_Cuda
  This function prints the content of a CUDA vector to outfile.
  void N_VPrintFile_Cuda(N_Vector v, FILE *outfile);

Notes

• When there is a need to access components of an N_Vector_Cuda, v, it is recommended 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. 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 NVECTOR_RAJA module can be utilized for single-node parallelism or in a distributed context with MPI. The header file to include when using this module for single-node parallelism is nvectorraja.h. The header file to include when using this module in the distributed case is nvector_mpiraja.h. Note that only the NVECTOR_RAJA constructor signature differs between the two header files. The installed module libraries to link to are libsundials_nvecraja.lib in the single-node case, or libsundials_nvecmpicudaraja.lib in the distributed case. Only one one of
these libraries may be linked to when creating an executable or library. SUNDIALS must be built with MPI support if the distributed library is desired. The extension, .lib, is typically .so for shared libraries and .a for static libraries.

Unlike other native SUNDIALS vector types, NVVECTOR_RAJA does not provide macros to access its member variables. Instead, user should use the accessor functions in the namespace sunrajavec.

- getDevData(N_Vector v)
  This function takes a N_Vector as an argument and returns a raw pointer to the vector data on the device (GPU). It is the user’s responsibility to ensure that the vector argument is of the correct N_Vector type.

- getHostData(N_Vector v)
  This function takes a N_Vector as an argument and returns a raw pointer to the vector data on the host (CPU memory). It is the user’s responsibility to ensure that the vector argument is of the correct N_Vector type.

- getSize(N_Vector v)
  Returns the vector’s local length.

- getGlobalSize(N_Vector v)
  Returns the vector’s global length.

- getMPIComm(N_Vector v)
  Takes a N_Vector as an argument and returns a sundials communicator of type SUNDIALS_Comm.

The NVVECTOR_RAJA module defines the implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4, except for N_VDotProdMulti, N_VWrmsNormVectorArray, and N_VWrmsNormMaskVectorArray as support for arrays of reduction vectors is not yet supported in RAJA. These function will be added to the NVVECTOR_RAJA implementation in the future. Additionally the vector operations N_VGetArrayPointer and N_VSetArrayPointer are not implemented by the RAJA vector. As such, this vector cannot be used with the SUNDIALS Fortran interfaces, nor with the SUNDIALS direct solvers and preconditioners. The NVVECTOR_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 NVVECTOR_RAJA are provided in some example programs for CVODE [19].

The names of vector operations are obtained from those in Tables 6.2, 6.3, and 6.4, by appending the suffix _Raja (e.g. N_VDestroy_Raja). The module NVVECTOR_RAJA provides the following additional user-callable routines:

- N_VNew_Raja
  
  *Note: this function signature is defined in the header nvvector_mpiraja.h and should be used when using this module in a distributed context.* This function creates and allocates memory for a RAJA N_Vector. The memory is allocated on both host and device. Its arguments are local and global vector lengths, as well as the MPI communicator. Use this constructor with the libsundials_nveccmpicudaraja.lib library.

  
  N_Vector N_VNew_Raja(MPI_Comm comm,
                       sunindextype local_length,
                       sunindextype global_length);

- N_VNew_Raja
  
  *Note: this function signature is defined in the header nvvector_raja.h and should be used when using this module for single-node parallelism.* This function creates and allocates memory for a RAJA N_Vector on a single node. The memory is allocated on both host and device. Its only argument is vector length. Use this constructor with the libsundials_nveccudadarajalib library.
N_Vector N_VNew_Raja(sunindextype 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 sunraja::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_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 RAJA vector to stdout.

void N_VPrint_Raja(N_Vector v);

- **N_VPrintFile_Raja**
  This function prints the content of a RAJA vector to outfile.

void N_VPrintFile_Raja(N_Vector v, FILE *outfile);

**Notes**

- When there is a need to access components of an N_Vector_Raja, v, it is recommended to use functions N_VGetDeviceArrayPointer_Raja or N_VGetHostArrayPointer_Raja.

- To maximize efficiency, vector operations in the NVVECTOR_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_NVScale: Case 3: negate: \( z = -x \)
• Test_NVScale: Case 4: combination: \( z = cx \)
• Test_NVAbs: Create absolute value of vector.
• Test_NVAddConst: add constant vector: \( z = c + x \)
• Test_NVDotProd: Calculate dot product of two vectors.
• Test_NVMaxNorm: Create vector with known values, find and validate the max norm.
• Test_NVWrmsNorm: Create vector of known values, find and validate the weighted root mean square.
• Test_NVWrmsNormMask: Create vector of known values, find and validate the weighted root mean square using all elements except one.
• Test_NVMin: Create vector, find and validate the min.
• Test_NVWL2Norm: Create vector, find and validate the weighted Euclidean L2 norm.
• Test_NVL1Norm: Create vector, find and validate the L1 norm.
• Test_NVCompare: Compare vector with constant returning and validating comparison vector.
• Test_NVInvTest: Test \( z[i] = 1 / x[i] \)
• Test_NVConstrMask: Test mask of vector \( x \) with vector \( c \).
• Test_NVMinQuotient: Fill two vectors with known values. Calculate and validate minimum quotient.
• Test_NVLinearCombination Case 1a: Test \( x = a x \)
• Test_NVLinearCombination Case 1b: Test \( z = a x \)
• Test_NVLinearCombination Case 2a: Test \( x = a x + b y \)
• Test_NVLinearCombination Case 2b: Test \( z = a x + b y \)
• Test_NVLinearCombination Case 3a: Test \( x = x + a y + b z \)
• Test_NVLinearCombination Case 3b: Test \( x = a x + b y + c z \)
• Test_NVLinearCombination Case 3c: Test \( w = a x + b y + c z \)
• Test_NVScaleAddMulti Case 1a: \( y = a x + y \)
• Test_NVScaleAddMulti Case 1b: \( z = a x + y \)
• Test_NVScaleAddMulti Case 2a: \( Y[i] = c[i] x + Y[i], i = 1,2,3 \)
• Test_NVScaleAddMulti Case 2b: \( Z[i] = c[i] x + Y[i], i = 1,2,3 \)
• Test_NVDotProdMulti Case 1: Calculate the dot product of two vectors
• Test_NVDotProdMulti Case 2: Calculate the dot product of one vector with three other vectors in a vector array.
• Test_NVLinearSumVectorArray Case 1: \( z = a x + b y \)
• Test_NVLinearSumVectorArray Case 2a: \( Z[i] = a X[i] + b Y[i] \)
• Test_NVLinearSumVectorArray Case 2b: \( X[i] = a X[i] + b Y[i] \)
- **Test_N_VLinearSumVectorArray** Case 2c: \( Y[i] = a \cdot X[i] + b \cdot Y[i] \)
- **Test_N_VScaleVectorArray** Case 1a: \( y = c \cdot y \)
- **Test_N_VScaleVectorArray** Case 1b: \( z = c \cdot y \)
- **Test_N_VScaleVectorArray** Case 2a: \( Y[i] = c[i] \cdot Y[i] \)
- **Test_N_VScaleVectorArray** Case 2b: \( Z[i] = c[i] \cdot Y[i] \)
- **Test_N_VScaleVectorArray** Case 1a: \( z = c \)
- **Test_N_VScaleVectorArray** Case 1b: \( Z[i] = c \)
- **Test_N_VWrmsNormVectorArray** Case 1a: Create a vector of known values, find and validate the weighted root mean square norm.
- **Test_N_VWrmsNormVectorArray** Case 1b: Create a vector array of three vectors of known values, find and validate the weighted root mean square norm of each.
- **Test_N_VWrmsNormMaskVectorArray** Case 1a: Create a vector of known values, find and validate the weighted root mean square norm using all elements except one.
- **Test_N_VWrmsNormMaskVectorArray** Case 1b: Create a vector array of three vectors of known values, find and validate the weighted root mean square norm of each using all elements except one.
- **Test_N_VScaleAddMultiVectorArray** Case 1a: \( y = a \cdot x + y \)
- **Test_N_VScaleAddMultiVectorArray** Case 1b: \( z = a \cdot x + y \)
- **Test_N_VScaleAddMultiVectorArray** Case 2a: \( Y[j][0] = a[j] \cdot X[0] + Y[j][0] \)
- **Test_N_VScaleAddMultiVectorArray** Case 2b: \( Z[j][0] = a[j] \cdot X[0] + Y[j][0] \)
- **Test_N_VScaleAddMultiVectorArray** Case 3a: \( Y[0][i] = a[0] \cdot X[i] + Y[0][i] \)
- **Test_N_VScaleAddMultiVectorArray** Case 3b: \( Z[0][i] = a[0] \cdot X[i] + Y[0][i] \)
- **Test_N_VScaleAddMultiVectorArray** Case 4a: \( Y[j][i] = a[j] \cdot X[i] + Y[j][i] \)
- **Test_N_VScaleAddMultiVectorArray** Case 4b: \( Z[j][i] = a[j] \cdot X[i] + Y[j][i] \)
- **Test_N_VLinearCombinationVectorArray** Case 1a: \( x = a \cdot x \)
- **Test_N_VLinearCombinationVectorArray** Case 1b: \( z = a \cdot x \)
- **Test_N_VLinearCombinationVectorArray** Case 2a: \( x = a \cdot x + b \cdot y \)
- **Test_N_VLinearCombinationVectorArray** Case 2b: \( z = a \cdot x + b \cdot y \)
- **Test_N_VLinearCombinationVectorArray** Case 3a: \( x = a \cdot x + b \cdot y + c \cdot z \)
- **Test_N_VLinearCombinationVectorArray** Case 3b: \( w = a \cdot x + b \cdot y + c \cdot z \)
- **Test_N_VLinearCombinationVectorArray** Case 4a: \( X[0][i] = c[0] \cdot X[0][i] \)
- **Test_N_VLinearCombinationVectorArray** Case 4b: \( Z[i] = c[0] \cdot X[0][i] \)
- **Test_N_VLinearCombinationVectorArray** Case 5a: \( X[0][i] = c[0] \cdot X[0][i] + c[1] \cdot X[1][i] \)
- **Test_N_VLinearCombinationVectorArray** Case 5b: \( Z[i] = c[0] \cdot X[0][i] + c[1] \cdot X[1][i] \)
- **Test_N_VLinearCombinationVectorArray** Case 6a: \( X[0][i] = c[0] \cdot X[0][i] + c[1] \cdot X[1][i] + c[2] \cdot X[2][i] \)
- **Test_N_VLinearCombinationVectorArray** Case 6b: \( X[0][i] = c[0] \cdot X[0][i] + c[1] \cdot X[1][i] + c[2] \cdot X[2][i] \)
- **Test_N_VLinearCombinationVectorArray** Case 6c: \( Z[i] = c[0] \cdot X[0][i] + c[1] \cdot X[1][i] + c[2] \cdot X[2][i] \)
6.10 NVECTOR functions used by KINSOL

In Table 6.5 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 KINLS stands for the generic 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 6.5: List of vector functions usage by KINSOL code modules

<table>
<thead>
<tr>
<th>Function</th>
<th>KINSOL</th>
<th>KINLS</th>
<th>KINBBDPRE</th>
<th>FKINSOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_VGetVectorID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VClone</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VCloneEmpty</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>N_VDestroy</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>N_VSpace</td>
<td>✓</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VGetArrayPointer</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>N_VSetArrayPointer</td>
<td>1</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VLinearSum</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VConst</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VProd</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VDiv</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VScale</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>N_VAbs</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VInv</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VDotProd</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VMaxNorm</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VMin</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VWl2Norm</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VLinNorm</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VConstrMask</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VMinQuotient</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VLinearCombination</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_VDotProdMulti</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Special cases (numbers match markings in table):

1. These routines are only required if an internal difference-quotient routine for constructing dense or band Jacobian matrices is used.

2. This routine is optional, and is only used in estimating space requirements for IDA modules for user feedback.

3. These routines are only required if the internal difference-quotient routine for approximating the Jacobian-vector product is used.

Each SUNLINSOL object may require additional NVECTOR routines not listed in the table above. Please see the relevant descriptions of these modules in Sections 8.3-8.13 for additional detail on their NVECTOR requirements.
The vector functions listed in Table 6.2 that are not used by KINSOL are \texttt{N_VAddConst}, \texttt{N_VWrmsNorm}, \texttt{N_VWrmsNormMask}, \texttt{N_VCompare}, and \texttt{N_VInvTest}. Therefore a user-supplied \texttt{NVECTOR} module for KINSOL could omit these functions.

The optional function \texttt{N_VLinearCombination} is only used when Anderson acceleration is enabled or the SPBCGS, SPTFQMR, SPGMR, or SPFGMR linear solvers are used. \texttt{N_VDotProd} is only used when Anderson acceleration is enabled or Classical Gram-Schmidt is used with SPGMR or SPFGMR. The remaining operations from Tables 6.3 and 6.4 are unused and a user-supplied \texttt{NVECTOR} module for KINSOL could omit these operations.
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.

<table>
<thead>
<tr>
<th>Matrix ID</th>
<th>Matrix type</th>
<th>ID Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNMATRIX_DENSE</td>
<td>Dense $M \times N$ matrix</td>
<td>0</td>
</tr>
<tr>
<td>SUNMATRIX_BAND</td>
<td>Band $M \times M$ matrix</td>
<td>1</td>
</tr>
<tr>
<td>SUNMATRIX_SPARSE</td>
<td>Sparse (CSR or CSC) $M \times N$ matrix</td>
<td>2</td>
</tr>
<tr>
<td>SUNMATRIX_CUSTOM</td>
<td>User-provided custom matrix</td>
<td>3</td>
</tr>
</tbody>
</table>

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:

```c
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNMatGetID</td>
<td>id = SUNMatGetID(A); Returns the type identifier for the matrix A. This is used to</td>
</tr>
<tr>
<td></td>
<td>determine the matrix implementation type (e.g., dense, banded, sparse, . . .) from</td>
</tr>
<tr>
<td></td>
<td>the abstract SUNMatrix interface. This is used to assess compatibility with SUNDIALS-</td>
</tr>
<tr>
<td></td>
<td>provided linear solver implementations. Returned values are given in the Table 7.1.</td>
</tr>
</tbody>
</table>
Name | Usage and Description
---|---
SUNMatClone | \( B = \text{SUNMatClone}(A); \)  
Creates a new SUNMatrix of the same type as an existing matrix \( A \) and sets the \( \text{ops} \) field. It does not copy the matrix, but rather allocates storage for the new matrix.
SUNMatDestroy | \( \text{SUNMatDestroy}(A); \)  
Destroys the SUNMatrix \( A \) and frees memory allocated for its internal data.
SUNMatSpace | \( \text{ier} = \text{SUNMatSpace}(A, &\text{lrw}, &\text{liw}); \)  
Returns the storage requirements for the matrix \( A \). \text{lrw} is a long int containing the number of realtype words and \( \text{liw} \) 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 | \( \text{ier} = \text{SUNMatZero}(A); \)  
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 | \( \text{ier} = \text{SUNMatCopy}(A,B); \)  
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 | \( \text{ier} = \text{SUNMatScaleAdd}(c, A, B); \)  
Performs the operation \( A = cA + B \). The return value is an integer flag denoting success/failure of the operation.
SUNMatScaleAddI | \( \text{ier} = \text{SUNMatScaleAddI}(c, A); \)  
Performs the operation \( A = cA + I \). The return value is an integer flag denoting success/failure of the operation.
SUNMatMatvec | \( \text{ier} = \text{SUNMatMatvec}(A, x, y); \)  
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.

<table>
<thead>
<tr>
<th>Matrix Interface</th>
<th>Serial</th>
<th>Parallel (MPI)</th>
<th>OpenMP</th>
<th>pThreads</th>
<th>hypre Vec.</th>
<th>PETSc Vec.</th>
<th>CUDA</th>
<th>RAJA</th>
<th>User Suppl.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

continued on next page
### 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:

```c
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 \(0 \leq i < M \text{ and } 0 \leq j < N\)) may be accessed via `data[j*M+i]`.
- **ldata** - length of the data array (\(= M \cdot N\)).
- **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 \(0 \leq i < M \text{ and } 0 \leq j < N\)) 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.

- **SMCONTENT_D**

  This macro gives access to the contents of the dense SUNMatrix.

  The assignment `A_cont = SMCONTENT_D(A)` sets `A_cont` to be a pointer to the dense SUNMatrix content structure.

  Implementation:

  ```c
  #define SMCONTENT_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 $A_{\text{rows}} = \text{SM}_\text{ROWS}_D(A)$ sets $A_{\text{rows}}$ to be the number of rows in the matrix $A$. Similarly, the assignment $\text{SM}_\text{COLUMNS}_D(A) = A_{\text{cols}}$ sets the number of columns in $A$ to equal $A_{\text{cols}}$.

Implementation:

```c
#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 $A_{\text{data}} = \text{SM}_\text{DATA}_D(A)$ sets $A_{\text{data}}$ to be a pointer to the first component of the data array for the dense SUNMatrix $A$. The assignment $\text{SM}_\text{DATA}_D(A) = A_{\text{data}}$ sets the data array of $A$ to be $A_{\text{data}}$ by storing the pointer $A_{\text{data}}$.

Similarly, the assignment $A_{\text{cols}} = \text{SM}_\text{COLS}_D(A)$ sets $A_{\text{cols}}$ to be a pointer to the array of column pointers for the dense SUNMatrix $A$. The assignment $\text{SM}_\text{COLS}_D(A) = A_{\text{cols}}$ sets the column pointer array of $A$ to be $A_{\text{cols}}$ by storing the pointer $A_{\text{cols}}$.

Implementation:

```c
#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 $\text{col}_j = \text{SM}_\text{COLUMN}_D(A,j)$ sets $\text{col}_j$ to be a pointer to the first entry of the $j$-th column of the $M \times N$ dense matrix $A$ (with $0 \leq j < N$). The type of the expression $\text{SM}_\text{COLUMN}_D(A,j)$ is realtype *. The pointer returned by the call $\text{SM}_\text{COLUMN}_D(A,j)$ can be treated as an array which is indexed from 0 to $M - 1$.

The assignments $\text{SM}_\text{ELEMENT}_D(A,i,j) = a_{ij}$ and $a_{ij} = \text{SM}_\text{ELEMENT}_D(A,i,j)$ reference the $(i,j)$-th element of the $M \times N$ dense matrix $A$ (with $0 \leq i < M$ and $0 \leq j < N$).

Implementation:

```c
#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.

  ```c
  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.

  ```c
  void SUNDenseMatrix_Print(SUNMatrix A, FILE* outfile);
  ```
Description of the SUNMatrix module

- **SUNDenseMatrix_Rows**
  This function returns the number of rows in the dense SUNMatrix.
  
  ```c
  sunindextype SUNDenseMatrix_Rows(SUNMatrix A);
  ```

- **SUNDenseMatrix_Columns**
  This function returns the number of columns in the dense SUNMatrix.
  
  ```c
  sunindextype SUNDenseMatrix_Columns(SUNMatrix A);
  ```

- **SUNDenseMatrix_LData**
  This function returns the length of the data array for the dense SUNMatrix.
  
  ```c
  sunindextype SUNDenseMatrix_LData(SUNMatrix A);
  ```

- **SUNDenseMatrix_Data**
  This function returns a pointer to the data array for the dense SUNMatrix.
  
  ```c
  realtype* SUNDenseMatrix_Data(SUNMatrix A);
  ```

- **SUNDenseMatrix_Cols**
  This function returns a pointer to the cols array for the dense SUNMatrix.
  
  ```c
  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 M - 1.
  
  ```c
  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 \( A_{\text{data}} = \text{SM\_DATA\_D}(A) \) or \( A_{\text{data}} = \text{SUNDenseMatrix\_Data}(A) \) and then access \( A_{\text{data}}[i] \) within the loop.
  
  - First obtain the array of column pointers via \( A_{\text{cols}} = \text{SM\_COLS\_D}(A) \) or \( A_{\text{cols}} = \text{SUNDenseMatrix\_Cols}(A) \), and then access \( A_{\text{cols}}[j][i] \) within the loop.
  
  - Within a loop over the columns, access the column pointer via \( A_{\text{colj}} = \text{SUNDenseMatrix\_Column}(A, j) \) and then to access the entries within that column using \( A_{\text{colj}}[i] \) within the loop.

  All three of these are more efficient than using \( \text{SM\_ELEMENT\_D}(A, i, j) \) 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 \( \text{FSUNDenseMatInit}(\text{code}, M, N, \text{ier}) \) 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 \( \text{FSUNDenseMassMatInit}(M, N, \text{ier}) \) 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:

```c
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+ml+1)
- **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·N)
- **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.
Description of the SUNMatrix module

Figure 7.1: Diagram of the storage for the sunmatrix band module. Here $A$ is an $N \times N$ band matrix with upper and lower half-bandwidths $\mu_u$ and $\mu_l$, respectively. The rows and columns of $A$ are numbered from 0 to $N - 1$ 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.
7.2 The SUNMatrix_Band implementation

- **SM_CONTENT_B**

  This routine gives access to the contents of the banded SUNMatrix.

  The assignment \( A_{\text{cont}} = \text{SM}_{\text{CONTENT}}_{\text{B}}(A) \) sets \( A_{\text{cont}} \) to be a pointer to the banded SUNMatrix content structure.

  Implementation:

  \[
  \text{#define SM}_{\text{CONTENT}}_{\text{B}}(A) \quad (\text{SUNMatrixContent}_{\text{Band}})(A->\text{content})
  \]

- **SM_ROWS_B, SM_COLUMNS_B, SM UBAND_B, SM LBAND_B, SM SUBAND_B, SM LDIM_B, and SM DATA_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 \( A_{\text{rows}} = \text{SM}_{\text{ROWS}}_{\text{B}}(A) \) sets \( A_{\text{rows}} \) to be the number of rows in the matrix \( A \).

  Similarly, the assignment \( \text{SM}_{\text{COLUMNS}}_{\text{B}}(A) = A_{\text{cols}} \) sets the number of columns in \( A \) to equal \( A_{\text{cols}} \).

  Implementation:

  \[
  \text{#define SM}_{\text{ROWS}}_{\text{B}}(A) \quad (\text{SM}_{\text{CONTENT}}_{\text{B}}(A)->M) \\
  \text{#define SM}_{\text{COLUMNS}}_{\text{B}}(A) \quad (\text{SM}_{\text{CONTENT}}_{\text{B}}(A)->N) \\
  \text{#define SM}_{\text{UBAND}}_{\text{B}}(A) \quad (\text{SM}_{\text{CONTENT}}_{\text{B}}(A)->\text{mu}) \\
  \text{#define SM}_{\text{LBAND}}_{\text{B}}(A) \quad (\text{SM}_{\text{CONTENT}}_{\text{B}}(A)->\text{ml}) \\
  \text{#define SM}_{\text{SUBAND}}_{\text{B}}(A) \quad (\text{SM}_{\text{CONTENT}}_{\text{B}}(A)->\text{s}_{\text{mu}}) \\
  \text{#define SM}_{\text{LDIM}}_{\text{B}}(A) \quad (\text{SM}_{\text{CONTENT}}_{\text{B}}(A)->\text{ldim}) \\
  \text{#define SM}_{\text{LDATA}}_{\text{B}}(A) \quad (\text{SM}_{\text{CONTENT}}_{\text{B}}(A)->\text{ldata})
  \]

- **SM_DATA_B and SM_COLS_B**

  These macros give access to the data and cols pointers for the matrix entries.

  The assignment \( A_{\text{data}} = \text{SM}_{\text{DATA}}_{\text{B}}(A) \) sets \( A_{\text{data}} \) to be a pointer to the first component of the data array for the banded SUNMatrix \( A \).

  The assignment \( \text{SM}_{\text{DATA}}_{\text{B}}(A) = A_{\text{data}} \) sets the data array of \( A \) to be \( A_{\text{data}} \) by storing the pointer \( A_{\text{data}} \).

  Similarly, the assignment \( A_{\text{cols}} = \text{SM}_{\text{COLS}}_{\text{B}}(A) \) sets \( A_{\text{cols}} \) to be a pointer to the array of column pointers for the banded SUNMatrix \( A \).

  The assignment \( \text{SM}_{\text{COLS}}_{\text{B}}(A) = A_{\text{cols}} \) sets the column pointer array of \( A \) to be \( A_{\text{cols}} \) by storing the pointer \( A_{\text{cols}} \).

  Implementation:

  \[
  \text{#define SM}_{\text{DATA}}_{\text{B}}(A) \quad (\text{SM}_{\text{CONTENT}}_{\text{B}}(A)->\text{data}) \\
  \text{#define SM}_{\text{COLS}}_{\text{B}}(A) \quad (\text{SM}_{\text{CONTENT}}_{\text{B}}(A)->\text{cols})
  \]

- **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 \( \text{SM}_{\text{ELEMENT}}_{\text{B}}(A,i,j) = a_{ij} \) and \( a_{ij} = \text{SM}_{\text{ELEMENT}}_{\text{B}}(A,i,j) \) reference the \((i,j)\)-th element of the \( N \times N \) band matrix \( A \), where \( 0 \leq i,j \leq N-1 \). The location \((i,j)\) should further satisfy \( j-\text{mu} \leq i \leq j+\text{ml} \).

  The assignment \( \text{col}_j = \text{SM}_{\text{COLUMN}}_{\text{B}}(A,j) \) sets \( \text{col}_j \) to be a pointer to the diagonal element of the \( j \)-th column of the \( N \times N \) band matrix \( A \), \( 0 \leq j \leq N-1 \). The type of the expression \( \text{SM}_{\text{COLUMN}}_{\text{B}}(A,j) \) is realtype *. The pointer returned by the call \( \text{SM}_{\text{COLUMN}}_{\text{B}}(A,j) \) can be treated as an array which is indexed from \( -\text{mu} \) to \( \text{ml} \).

  The assignments \( \text{SM}_{\text{COLUMN ELEMENT}}_{\text{B}}(\text{col}_j,i,j) = a_{ij} \) and \( a_{ij} = \text{SM}_{\text{COLUMN ELEMENT}}_{\text{B}}(\text{col}_j,i,j) \) reference the \((i,j)\)-th entry of the band matrix \( A \) when used in conjunction with \( \text{SM}_{\text{COLUMN}}_{\text{B}} \) to reference the \( j \)-th column through \( \text{col}_j \). The index \((i,j)\) should satisfy \( j-\text{mu} \leq i \leq j+\text{ml} \).
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, this value should be
  
  - at least min(N-1,mu+ml) if the matrix will be used by the SUNLINSOL_BAND module;
  - exactly equal to mu+ml if the matrix will be used by the SUNLINSOL_LAPACKBAND module;
  - at least mu if used in some other manner.

  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);
7.3 The SUNMatrix_Sparse implementation

- **SUNBandMatrix_LDim**
  This function returns the length of the leading dimension of the banded SUNMatrix.
  
  ```c
  sunindextype SUNBandMatrix_LDim(SUNMatrix A);
  ```

- **SUNBandMatrix_Data**
  This function returns a pointer to the data array for the banded SUNMatrix.
  
  ```c
  realtype* SUNBandMatrix_Data(SUNMatrix A);
  ```

- **SUNBandMatrix_Cols**
  This function returns a pointer to the cols array for the banded SUNMatrix.
  
  ```c
  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 $\mu$.
  
  ```c
  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 $A.cols = SM_COL_B(A)$ 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 SUONDIALS, 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 SUONDIALS 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 SUONDIALS, 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 on 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 $5 \times 4$ 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
\[
\begin{align*}
M &= 5; \\
N &= 4; \\
NNZ &= 8; \\
NP &= N; \\
data &= \{3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0\}; \\
sparsetype &= \text{CSC}_\text{MAT}; \\
indexvals &= \{1, 3, 0, 2, 0, 1, 3, 4\}; \\
indexptrs &= \{0, 2, 4, 5, 8\};
\end{align*}
\]
or
\[
\begin{align*}
M &= 5; \\
N &= 4; \\
NNZ &= 10; \\
NP &= N; \\
data &= \{3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0, *, *\}; \\
sparsetype &= \text{CSC}_\text{MAT}; \\
indexvals &= \{1, 3, 0, 2, 0, 1, 3, 4, *, *\}; \\
indexptrs &= \{0, 2, 4, 5, 8\};
\end{align*}
\]
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 \text{indexptrs} is 8, indicating the total number of nonzero entries in the matrix.

Similarly, in CSR format, the same matrix could be stored as
\[
\begin{align*}
M &= 5; \\
N &= 4; \\
NNZ &= 8; \\
NP &= N; \\
data &= \{3.0, 1.0, 3.0, 2.0, 7.0, 1.0, 9.0, 5.0\}; \\
sparsetype &= \text{CSR}_\text{MAT}; \\
indexvals &= \{1, 2, 0, 3, 1, 0, 3, 3\}; \\
indexptrs &= \{0, 2, 4, 5, 7, 8\};
\end{align*}
\]
The header file to include when using this module is \texttt{sunmatrix/sunmatrix_sparse.h}. The \texttt{SUNMatrix_Sparse} module is accessible from all \texttt{sundials} solvers \textit{without} linking to the \texttt{libsundials_sunmatrixsparse} module library.

The following macros are provided to access the content of a \texttt{SUNMatrix_Sparse} matrix. The prefix \texttt{SM} in the names denotes that these macros are for \texttt{SUNMatrix} implementations, and the suffix \texttt{S} denotes that these are specific to the \textit{sparse} version.

- \texttt{SM\_CONTENT\_S}
  
  This routine gives access to the contents of the sparse \texttt{SUNMatrix}.
  
  The assignment $A\_\text{cont} = \text{SM\_CONTENT\_S}(A)$ sets $A\_\text{cont}$ to be a pointer to the sparse \texttt{SUNMatrix} content structure.
  
  Implementation:
  
  \[
  \text{#define } \text{SM\_CONTENT\_S}(A) \quad \left( (\text{SUNMatrixContent\_Sparse})(A->\text{content}) \right)
  \]
Figure 7.2: Diagram of the storage for a compressed-sparse-column matrix. Here A is an $M \times N$ 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 $M - 1$, 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 $N + 1$ 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.
7.3 The SUNMatrix_Sparse implementation

- **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.
  
  These may be used either to retrieve or to set these values. For example, the assignment $A_{\text{rows}} = \text{SM_ROWS}_S(A)$ sets $A_{\text{rows}}$ to be the number of rows in the matrix $A$. Similarly, the assignment $\text{SM_COLUMNS}_S(A) = A_{\text{cols}}$ sets the number of columns in $A$ to equal $A_{\text{cols}}$.

  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 $A_{\text{data}} = \text{SM_DATA}_S(A)$ sets $A_{\text{data}}$ to be a pointer to the first component of the data array for the sparse SUNMatrix $A$. The assignment $\text{SM_DATA}_S(A) = A_{\text{data}}$ sets the data array of $A$ to be $A_{\text{data}}$ by storing the pointer $A_{\text{data}}$.
  
  Similarly, the assignment $A_{\text{indexvals}} = \text{SM_INDEXVALS}_S(A)$ sets $A_{\text{indexvals}}$ 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 $A_{\text{indexptrs}} = \text{SM_INDEXPTRS}_S(A)$ sets $A_{\text{indexptrs}}$ 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 $\text{sparsetype}$ indicating whether to use CSR or CSC format (valid arguments are $\text{CSR\_MAT}$ or $\text{CSC\_MAT}$).

  ```
  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 $\text{droptol}$ into the sparse matrix structure.

  Requirements:
  
  - $A$ must have type $\text{SUNMATRIX\_DENSE}$;
- \texttt{droptol} must be non-negative;
- \texttt{sparsetype} must be either \texttt{CSC\_MAT} or \texttt{CSR\_MAT}.

The function returns NULL if any requirements are violated, or if the matrix storage request cannot be satisfied.

\begin{verbatim}
SUNMatrix SUNSparseFromDenseMatrix(SUNMatrix A, realtype droptol,
    int sparsetype);
\end{verbatim}

- \textbf{SUNSparseFromBandMatrix}

This function creates a new sparse matrix from an existing band matrix by copying all values with magnitude larger than \texttt{droptol} into the sparse matrix structure.

Requirements:
- \texttt{A} must have type \texttt{SUNMATRIX\_BAND};
- \texttt{droptol} must be non-negative;
- \texttt{sparsetype} must be either \texttt{CSC\_MAT} or \texttt{CSR\_MAT}.

The function returns NULL if any requirements are violated, or if the matrix storage request cannot be satisfied.

\begin{verbatim}
SUNMatrix SUNSparseFromBandMatrix(SUNMatrix A, realtype droptol,
    int sparsetype);
\end{verbatim}

- \textbf{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, \texttt{indexptrs[NP]}). Returns 0 on success and 1 on failure (e.g. if the input matrix is not sparse).

\begin{verbatim}
int SUNSparseMatrix\_Realloc(SUNMatrix A);
\end{verbatim}

- \textbf{SUNSparseMatrix\_Reallocate}

This function reallocates internal storage arrays in a sparse matrix so that the resulting sparse matrix has storage for a specified number of nonzeros. Returns 0 on success and 1 on failure (e.g. if the input matrix is not sparse or if \texttt{NNZ} is negative).

\begin{verbatim}
int SUNSparseMatrix\_Reallocate(SUNMatrix A, sunindextype NNZ);
\end{verbatim}

- \textbf{SUNSparseMatrix\_Print}

This function prints the content of a sparse SUNMatrix to the output stream specified by \texttt{outfile}. Note: \texttt{stdout} or \texttt{stderr} may be used as arguments for \texttt{outfile} to print directly to standard output or standard error, respectively.

\begin{verbatim}
void SUNSparseMatrix\_Print(SUNMatrix A, FILE* outfile);
\end{verbatim}

- \textbf{SUNSparseMatrix\_Rows}

This function returns the number of rows in the sparse SUNMatrix.

\begin{verbatim}
sunindextype SUNSparseMatrix\_Rows(SUNMatrix A);
\end{verbatim}

- \textbf{SUNSparseMatrix\_Columns}

This function returns the number of columns in the sparse SUNMatrix.

\begin{verbatim}
sunindextype SUNSparseMatrix\_Columns(SUNMatrix A);
\end{verbatim}
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:

- **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.
Description of the SUNMatrix module

- **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_SUNMatDestroy**: 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 SUNMatScaleAdd, 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. The main KINSOL integrator does not call any SUNMATRIX functions directly, so the table columns are specific to the KINLS interface and the KINBBDPRE preconditioner module. We further note that the KINLS interface only utilizes these routines when supplied with a matrix-based linear solver, i.e., the SUNMATRIX object passed to KINSetLinearSolver was not NULL.

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>
<thead>
<tr>
<th>Function</th>
<th>KINLS</th>
<th>KINBBDPRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNMatGetID</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SUNMatDestroy</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SUNMatZero</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SUNMatSpace</td>
<td></td>
<td>†</td>
</tr>
</tbody>
</table>

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.

We note that the KINBBDPRE preconditioner module is hard-coded to use the SUNDIALS-supplied band SUNMATRIX type, so the most useful information above for user-supplied SUNMATRIX implementations is the column relating the KINLS requirements.
Chapter 8

Description of the SUNLinearSolver module

For problems that involve the solution of linear systems of equations, the SUNDIALS packages operate using generic linear solver modules defined through the SUNLINSOL API. This allows SUNDIALS packages to utilize any valid SUNLINSOL implementation that provides a set of required functions. These functions can be divided into three categories. The first are the core linear solver functions. The second group consists of “set” routines to supply the linear solver object with functions provided by the SUNDIALS package, or for modification of solver parameters. The last group consists of “get” routines for retrieving artifacts (statistics, residual vectors, etc.) from the linear solver. All of these functions are defined in the header file sundials/sundials_linearsolver.h.

The implementations provided with SUNDIALS 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 matrix-free iterative methods. Moreover, advanced users can provide a customized SUNLinearSolver implementation to any SUNDIALS package, particularly in cases where they provide their own NVECTOR and/or SUNMATRIX modules.

Historically, the SUNDIALS packages have been designed to specifically leverage the use of either direct linear solvers or matrix-free, scaled, preconditioned, iterative linear solvers. However, matrix-based iterative linear solvers are supported.

The iterative linear solvers packaged with SUNDIALS 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, these apply the underlying iterative algorithm to the transformed system

$$\tilde{A}\tilde{x} = \tilde{b}$$

where

$$\tilde{A} = S_1 P_1^{-1} A P_2^{-1} S_2^{-1},$$

$$\tilde{b} = S_1 P_1^{-1} b,$$

$$\tilde{x} = S_2 P_2 x,$$

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 $P_2 x$. 
SUNDIALS packages request that iterative linear solvers stop based on the 2-norm of the scaled preconditioned residual meeting a prescribed tolerance

$$\| \tilde{b} - \tilde{A}\tilde{x} \|_2 < \text{tol}. $$

When provided an iterative SUNLINSOL implementation that does not support the scaling matrices $S_1$ and $S_2$, SUNDIALS’ packages will adjust the value of tol accordingly. In this case, they instead request that iterative linear solvers stop based on the criteria

$$\| P_1^{-1}b - P_1^{-1}Ax \|_2 < \text{tol}. $$

We note that the corresponding adjustments to tol in this case are non-optimal, in that they cannot balance error between specific entries of the solution $x$, only the aggregate error in the overall solution vector.

We further note that not all of the SUNDIALS-provided iterative linear solvers support the full range of the above options (e.g., separate left/right preconditioning), and that some of the SUNDIALS packages only utilize a subset of these options. Further details on these exceptions are described in the documentation for each SUNLINSOL implementation, or for each SUNDIALS package.

## 8.0.1 SUNLinearSolver core functions

The core linear solver functions consist of four required routines to get the linear solver type (SUNLinSolGetType), initialize the linear solver object once all solver-specific options have been set (SUNLinSolInitialize), set up the linear solver object to utilize an updated matrix $A$ (SUNLinSolSetup), and solve the linear system $Ax = b$ (SUNLinSolSolve). The remaining routine for destruction of the linear solver object (SUNLinSolFree) is optional.

### SUNLinSolGetType

**Call**

```c
int type = SUNLinSolGetType(LS);
```

**Description**
The required function SUNLinSolGetType returns the type identifier for the linear solver LS. It is used to determine the solver type (direct or iterative) from the abstract SUNLinearSolver interface. This is used to assess compatibility with SUNDIALS-provided linear solver interfaces.

**Arguments**

- LS (SUNLinearSolver) a SUNLINSOL object.

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

- SUNNONLINEARSOLVER_DIRECT 0, the SUNLINSOL module uses direct methods to solve the linear system.
- SUNNONLINEARSOLVER_ITERATIVE 1, the SUNLINSOL module iteratively solves the linear system, stopping when the linear residual is within a prescribed tolerance.

**Notes**

### SUNLinSolInitialize

**Call**

```c
int retval = SUNLinSolInitialize(LS);
```

**Description**
The required function SUNLinSolInitialize performs linear solver initialization (assumes that all solver-specific options have been set).

**Arguments**

- LS (SUNLinearSolver) a SUNLINSOL object.

**Return value**
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 8.1.

**Notes**
**SUNLinSolSetup**

Call

```c
retval = SUNLinSolSetup(LS, A);
```

Description The `required` function `SUNLinSolSetup` 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.

Arguments
- `LS` (SUNLinearSolver) a SUNLINSOL object.
- `A` (SUNMatrix) a SUNMATRIX object.

Return value 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 8.1.

Notes

**SUNLinSolSolve**

Call

```c
retval = SUNLinSolSolve(LS, A, x, b, tol);
```

Description The `required` function `SUNLinSolSolve` solves a linear system \( Ax = b \).

Arguments
- `LS` (SUNLinearSolver) a SUNLINSOL object.
- `A` (SUNMatrix) a SUNMATRIX object.
- `x` (N_Vector) a NVECTOR object.
- `b` (N_Vector) a NVECTOR object.
- `tol` (realtype) the desired linear solver tolerance.

Return value 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 8.1.

Notes

- **Direct solvers**: can ignore the `*tol*` argument.
- **Matrix-free 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`.
- **Iterative solvers**: These should attempt to solve to the specified tolerance `tol` in a weighted 2-norm. If the solver does not support scaling then it should just use a 2-norm.

**SUNLinSolFree**

Call

```c
retval = SUNLinSolFree(LS);
```

Description The `optional` function `SUNLinSolFree` frees memory allocated by the linear solver.

Arguments `LS` (SUNLinearSolver) a SUNLINSOL object.

Return value This should return zero for a successful call and a negative value for a failure.

Notes

8.0.2 SUNLinearSolver set functions

The following set functions are used to supply linear solver modules with functions defined by the SUNDIALS packages and to modify solver parameters. Only the routine for setting the matrix-vector product routine is required, and that is only for matrix-free linear solver modules. Otherwise, all other set functions are optional. SUNLINSOL implementations that do not provide the functionality for any optional routine should leave the corresponding function pointer `NULL` instead of supplying a dummy routine.
Description of the SUNLinearSolver module

**SUNLinSolSetATimes**

**Call**
\[
\text{retval} = \text{SUNLinSolSetATimes}(\text{LS}, \ A\text{data}, \ A\text{Times});
\]

**Description**
The function **SUNLinSolSetATimes** is required for matrix-free linear solvers; otherwise it is optional.

This routine provides an **ATimesFn** function pointer, as well as a **void** * pointer to a data structure used by this routine, to a linear solver object. **SUNDIALS** packages 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.

**Arguments**
- **LS** (SUNLinearSolver) a SUNLINSOL object.
- **A\_data** (void*) data structure passed to **ATimes**.
- **ATimes** (ATimesFn) function pointer implementing the matrix-vector product routine.

**Return value**
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 8.1.

**Notes**

**SUNLinSolSetPreconditioner**

**Call**
\[
\text{retval} = \text{SUNLinSolSetPreconditioner}(\text{LS}, \ P\text{data}, \ P\text{set}, \ Psol);
\]

**Description**
The optional function **SUNLinSolSetPreconditioner** 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** package, which will provide translation between the generic **Pset** and **Psol** calls and the package- or user-supplied routines.

**Arguments**
- **LS** (SUNLinearSolver) a SUNLINSOL object.
- **P\_data** (void*) data structure passed to both **Pset** and **Psol**.
- **Pset** (PSetupFn) function pointer implementing the preconditioner setup.
- **Psol** (PSolveFn) function pointer implementing the preconditioner solve.

**Return value**
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 8.1.

**Notes**

**SUNLinSolSetScalingVectors**

**Call**
\[
\text{retval} = \text{SUNLinSolSetScalingVectors}(\text{LS}, \ s1, \ s2);
\]

**Description**
The optional function **SUNLinSolSetScalingVectors** provides left/right scaling vectors for the linear system solve. Here, **s1** and **s2** are NVECTOR of positive scale factors containing the diagonal of the matrices \(S_1\) and \(S_2\) from equations (8.1)-(8.2), respectively. Neither of these vectors need to be tested for positivity, and a NULL argument for either indicates that the corresponding scaling matrix is the identity.

**Arguments**
- **LS** (SUNLinearSolver) a SUNLINSOL object.
- **s1** (N\_Vector) diagonal of the matrix \(S_1\)
- **s2** (N\_Vector) diagonal of the matrix \(S_2\)

**Return value**
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 8.1.

**Notes**

**8.0.3 SUNLinearSolver get functions**

The following get functions allow **SUNDIALS** packages to retrieve results from the linear solve. All routines are optional.
**SUNLinSolNumIters**

Call

```c
its = SUNLinSolNumIters(LS);
```

Description The *optional* function `SUNLinSolNumIters` should return the number of linear iterations performed in the last ‘solve’ call.

Arguments

- `LS` (SUNLinearSolver) a SUNLINSOL object.

Return value

- `int` containing the number of iterations

Notes

**SUNLinSolResNorm**

Call

```c
rnorm = SUNLinSolResNorm(LS);
```

Description The *optional* function `SUNLinSolResNorm` should return the final residual norm from the last ‘solve’ call.

Arguments

- `LS` (SUNLinearSolver) a SUNLINSOL object.

Return value

- `realtype` containing the final residual norm

Notes

**SUNLinSolResid**

Call

```c
rvec = SUNLinSolResid(LS);
```

Description 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 *optional* routine may be called by the SUNDIALS package. This routine should return the NVECTOR containing the preconditioned initial residual vector.

Arguments

- `LS` (SUNLinearSolver) a SUNLINSOL object.

Return value

- `N_Vector` containing the final residual vector

Notes

Since `N_Vector` is actually a pointer, and the results are not modified, this routine should *not* require additional memory allocation. If the SUNLINSOL object does not retain a vector for this purpose, then this function pointer should be left NULL in the implementation.

**SUNLinSolLastFlag**

Call

```c
lflag = SUNLinSolLastFlag(LS);
```

Description The *optional* function `SUNLinSolLastFlag` should return the last error flag encountered within the linear solver. This is not called by the SUNDIALS packages directly; it allows the user to investigate linear solver issues after a failed solve.

Arguments

- `LS` (SUNLinearSolver) a SUNLINSOL object.

Return value

- `long int` containing the most recent error flag

Notes

**SUNLinSolSpace**

Call

```c
retval = SUNLinSolSpace(LS, &lrw, &liw);
```

Description The *optional* function `SUNLinSolSpace` should return the storage requirements for the linear solver `LS`.

Arguments

- `LS` (SUNLinearSolver) a SUNLINSOL object.
lrw (long int*) the number of realtype words stored by the linear solver.
lw (long int*) the number of integer words stored by the linear solver.

Return value 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 8.1.

Notes This function is advisory only, for use in determining a user’s total space requirements.

8.0.4 Functions provided by SUNDIALS packages

To interface with the SUNLINSOL modules, the SUNDIALS packages supply a variety of routines for evaluating the matrix-vector product, and setting up and applying the preconditioner. These package-provided routines translate between the user-supplied ODE, DAE or nonlinear systems and the generic interfaces to the linear systems of equations that result in their solution. The types for functions provided to a SUNLINSOL module are defined in the header file sundials/sundials_iterative.h, and are described below.

\textbf{ATimesFn}

Definition \texttt{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 \( z = A v \). Memory for \( z \) should already be allocated prior to calling this function. The vector \( v \) should be left unchanged.

Arguments \( A_{\text{data}} \) is a pointer to client data, the same as that supplied to \texttt{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

\textbf{PSetupFn}

Definition \texttt{typedef int (*PSetupFn)(void *P_data)}

Purpose These functions set up any requisite problem data in preparation for calls to the corresponding \texttt{PSolveFn}.

Arguments \( P\_{\text{data}} \) is a pointer to client data, the same pointer as that supplied to the routine \texttt{SUNLinSolSetPreconditioner}.

Return value This routine should return 0 if successful and a non-zero value if unsuccessful.

Notes

\textbf{PSolveFn}

Definition \texttt{typedef int (*PSolveFn)(void *P_data, N_Vector r, N_Vector z, realtype tol, int lr)}

Purpose These functions solve the preconditioner equation \( P z = r \) for the vector \( z \). Memory for \( z \) should already be allocated prior to calling this function. The parameter \( P\_{\text{data}} \) is a pointer to any information about \( P \) which the function needs in order to do its job (set up by the corresponding \texttt{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

\[ \| P z - r \|_{\text{W RMS}} < tol \]
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 \texttt{PSolveFn}.

Arguments

\texttt{P_data} is a pointer to client data, the same pointer as that supplied to the routine \texttt{SUNLinSolSetPreconditioner}.

\( r \) is the right-hand side vector for the preconditioner system

\( z \) is the solution vector for the preconditioner system

\( \text{tol} \) is the desired tolerance for an iterative preconditioner

\( \text{lr} \) 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

8.0.5 \texttt{SUNLinearSolver} return codes

The functions provided to \texttt{SUNLinearSolver} modules by each \texttt{SUNDIALS} package, and functions within the \texttt{SUNDIALS}-provided \texttt{SUNLinearSolver} implementations utilize a common set of return codes, shown in the Table 8.1. These adhere to a common pattern: 0 indicates success, a positive value corresponds to a recoverable failure, and a negative value indicates a non-recoverable failure. Aside from this pattern, the actual values of each error code are primarily to provide additional information to the user in case of a linear solver failure.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNLS_SUCCESS</td>
<td>0</td>
<td>successful call or converged solve</td>
</tr>
<tr>
<td>SUNLS_MEM_NULL</td>
<td>-1</td>
<td>the memory argument to the function is NULL</td>
</tr>
<tr>
<td>SUNLS_ILL_INPUT</td>
<td>-2</td>
<td>an illegal input has been provided to the function</td>
</tr>
<tr>
<td>SUNLS_MEM_FAIL</td>
<td>-3</td>
<td>failed memory access or allocation</td>
</tr>
<tr>
<td>SUNLS_ATIMES_FAIL_UNREC</td>
<td>-4</td>
<td>an unrecoverable failure occurred in the \texttt{ATimes} routine</td>
</tr>
<tr>
<td>SUNLS_PSET_FAIL_UNREC</td>
<td>-5</td>
<td>an unrecoverable failure occurred in the \texttt{Pset} routine</td>
</tr>
<tr>
<td>SUNLS_PSOLVE_FAIL_UNREC</td>
<td>-6</td>
<td>an unrecoverable failure occurred in the \texttt{Psolve} routine</td>
</tr>
<tr>
<td>SUNLS_PACKAGE_FAIL_UNREC</td>
<td>-7</td>
<td>an unrecoverable failure occurred in an external linear solver package</td>
</tr>
<tr>
<td>SUNLS_GS_FAIL</td>
<td>-8</td>
<td>a failure occurred during Gram-Schmidt orthogonalization</td>
</tr>
<tr>
<td>SUNLS_QRSOL_FAIL</td>
<td>-9</td>
<td>a singular ( R ) matrix was encountered in a QR factorization</td>
</tr>
</tbody>
</table>

\textit{continued on next page}
<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNLS_RES_REDUCED</td>
<td>1</td>
<td>an iterative solver reduced the residual, but did not converge to the desired tolerance</td>
</tr>
<tr>
<td>SUNLS_CONV_FAIL</td>
<td>2</td>
<td>an iterative solver did not converge (and the residual was not reduced)</td>
</tr>
<tr>
<td>SUNLS_ATIMES_FAIL_REC</td>
<td>3</td>
<td>a recoverable failure occurred in the ATimes routine</td>
</tr>
<tr>
<td>SUNLS_PSET_FAIL_REC</td>
<td>4</td>
<td>a recoverable failure occurred in the Pset routine</td>
</tr>
<tr>
<td>SUNLS_PSOOLVE_FAIL_REC</td>
<td>5</td>
<td>a recoverable failure occurred in the Psolve routine</td>
</tr>
<tr>
<td>SUNLS_PACKAGE_FAIL_REC</td>
<td>6</td>
<td>a recoverable failure occurred in an external linear solver package</td>
</tr>
<tr>
<td>SUNLS_QRFACRT_FAIL</td>
<td>7</td>
<td>a singular matrix was encountered during a QR factorization (sunlinsol_spgmr/sunlinsol_spfgmr)</td>
</tr>
<tr>
<td>SUNLS_LUFACRT_FAIL</td>
<td>8</td>
<td>a singular matrix was encountered during a LU factorization (sunlinsol_dense/sunlinsol_band)</td>
</tr>
</tbody>
</table>

8.0.6 The generic SUNLinearSolver module

SUNDIALS packages interact with specific SUNLINSOL implementations through the generic SUNLINSOL module on which all other SUNLINSOL implementations are built. The SUNLinearSolver type is a pointer to a structure containing an implementation-dependent content field, and an ops field. The type SUNLinearSolver is defined as

typedef struct _generic_SUNLinearSolver *SUNLinearSolver;

struct _generic_SUNLinearSolver {
    void *content;
    struct _generic_SUNLinearSolver_Ops *ops;
};

where the _generic_SUNLinearSolver_Ops structure is a list of pointers to the various actual linear solver operations provided by a specific implementation. The _generic_SUNLinearSolver_Ops structure 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 defined in Sections 8.0.1-8.0.3. These routines are in fact only wrappers to 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:

```c
int SUNLinSolInitialize(SUNLinearSolver S)
{
    return ((int) S->ops->initialize(S));
}
```

### 8.1 Compatibility of SUNLinearSolver modules

We note that not all SUNLINSOL types are compatible with all SUNMATRIX and NVECTOR types provided with SUNDIALS. In Table 8.2 we show the matrix-based 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>
<thead>
<tr>
<th>Linear Solver Interface</th>
<th>Dense Matrix</th>
<th>Banded Matrix</th>
<th>Sparse Matrix</th>
<th>User Supplied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Band</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>LapackDense</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>LapackBand</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>KLU</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SUPERLUMT</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>User supplied</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### 8.2 Implementing a custom SUNLinearSolver module

A particular implementation of the SUNLINSOL module must:

- 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.

We note that the function pointers for all unsupported optional routines should be set to NULL in the *ops* structure. This allows the SUNDIALS package that is using the SUNLINSOL object to know that the associated functionality is not supported.

Additionally, a SUNLINSOL implementation may do the following:
• Define and implement additional user-callable “set” routines acting on the SUNLinearSolver, e.g., for setting various configuration options to tune the linear solver to a particular problem.
• Provide additional user-callable “get” routines acting on the SUNLinearSolver object, e.g., for returning various solve statistics.

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 NVVECTOR implementations (NVVECTOR_SERIAL, NVVECTOR_OPENMP or NVVECTOR_PTHREADS).

8.3.1 SUNLINSOL_DENSE usage

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 module SUNLINSOL_DENSE provides the following user-callable constructor routine:

```
SUNLinSol_Dense Call LS = SUNLinSol_Dense(y, A);
```

Description The function SUNLinSol_Dense creates and allocates memory for a dense SUNLinearSolver object.

Arguments y (N_Vector) a template for cloning vectors needed within the solver
A (SUNMatrix) a SUNMATRIX_DENSE matrix template for cloning matrices needed within the solver

Return value This returns a SUNLinearSolver object. If either A or y are incompatible then this routine will return NULL.

Notes This routine will perform consistency checks to ensure that it is called with consistent NVVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX_DENSE matrix type and the NVVECTOR_SERIAL, NVVECTOR_OPENMP, and NVVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

For backwards compatibility, we also provide the wrapper function,

• SUNDenseLinearSolver

Wrapper function for SUNLinSol_Dense, with identical input and output arguments.

For solvers that include a Fortran interface module, the SUNLINSOL_DENSE module also includes a Fortran-callable function for creating a SUNLinearSolver object.

```
FSUNDENSELINSOLINIT Call FSUNDENSELINSOLINIT(code, ier)
```

Description The function FSUNDENSELINSOLINIT can be called for Fortran programs to create a dense SUNLinearSolver object.

Arguments code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).

Return value ier is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
8.3 The SUNLinearSolver_Dense implementation

Notes 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 SUNLINSOL_DENSE module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

FSUNMASSDENSELINSOLINIT
Call FSUNMASSDENSELINSOLINIT(ier)
Description The function FSUNMASSDENSELINSOLINIT can be called for Fortran programs to create a dense SUNLinearSolver object for mass matrix linear systems.

Arguments
Return value ier is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called after both the NVECTOR and SUNMATRIX mass-matrix objects have been initialized.

8.3.2 SUNLINSOL_DENSE description

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 (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 (O(N^2) cost).

The SUNLINSOL_DENSE module defines dense implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- 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
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).

8.4.1 SUNLINSOL_BAND usage

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 module SUNLINSOL_BAND provides the following user-callable constructor routine:

```c
SUNLinSol_Band
Call LS = SUNLinSol_Band(y, A);
Description The function SUNLinSol_Band creates and allocates memory for a band SUNLinearSolver object.
Arguments y (N_Vector) a template for cloning vectors needed within the solver
A (SUNMatrix) a SUNMATRIX_BAND matrix template for cloning matrices needed within the solver
Return value This returns a SUNLinearSolver object. If either A or y are incompatible then this routine will return NULL.
Notes 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.
Additionally, this routine will verify that the input matrix A is allocated with appropriate upper bandwidth storage for the LU factorization.
```

For backwards compatibility, we also provide the wrapper functions:

- **SUNBandLinearSolver**

Wrapper function for SUNLinSol_Band, with identical input and output arguments.

For solvers that include a Fortran interface module, the SUNLINSOL_BAND module also includes a Fortran-callable function for creating a SUNLinearSolver object.

```fortran
FSUNBANDLINSOLINIT
Call FSUNBANDLINSOLINIT(code, ier)
Description The function FSUNBANDLINSOLINIT can be called for Fortran programs to create a band SUNLinearSolver object.
Arguments code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
Return value ier is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes 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 SUNLINSOL_BAND module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.
```
8.4 The SUNLinearSolver\_Band implementation

### FSUNMASSBANDLINSOLINIT

**Call**

FSUNMASSBANDLINSOLINIT(ier)

**Description**
The function FSUNMASSBANDLINSOLINIT can be called for Fortran programs to create a band SUNLinearSolver object for mass matrix linear systems.

**Arguments**

- **Return value** `ier` is a `int` return completion flag equal to 0 for a success return and ~1 otherwise. See printed message for details in case of failure.

**Notes**
This routine must be called after both the nvector and sunmatrix mass-matrix objects have been initialized.

#### 8.4.2 SUNLINSOL\_BAND description

The SUNLINSOL\_BAND module defines the `content` field of a SUNLinearSolver to be the following structure:

```c
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 $m_l$, then the upper triangular factor $U$ can have upper bandwidth as big as $smu = \text{MIN}(N-1,\mu+m_l)$. The lower triangular factor $L$ has lower bandwidth $m_l$.

The SUNLINSOL\_BAND module defines band implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- 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
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 NVVECTOR implementations (NVVECTOR_SERIAL, NVVECTOR_OPENMP, or NVVECTOR_PTHREADS).

8.5.1 SUNLINSOL_LAPACKDENSE usage

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 module SUNLINSOL_LAPACKDENSE provides the following user-callable constructor routine:

```c
SUNLinSolLapackDense LS = SUNLinSolLapackDense(y, A);
```

Description
The function SUNLinSolLapackDense creates and allocates memory for a LAPACK-based, dense SUNLinearSolver object.

Arguments
- `y` (N_Vector) a template for cloning vectors needed within the solver
- `A` (SUNMatrix) a SUNMATRIX_DENSE matrix template for cloning matrices needed within the solver

Return value
This returns a SUNLinearSolver object. If either `A` or `y` are incompatible then this routine will return NULL.

Notes
This routine will perform consistency checks to ensure that it is called with consistent NVVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX_DENSE matrix type and the NVVECTOR_SERIAL, NVVECTOR_OPENMP, and NVVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

For backwards compatibility, we also provide the wrapper function,

- **SUNLapackDense**
  
  Wrapper function for SUNLinSolLapackDense, with identical input and output arguments.

For solvers that include a Fortran interface module, the SUNLINSOL_LAPACKDENSE module also includes a Fortran-callable function for creating a SUNLinearSolver object.

```c
FSUNLAPACKDENSEINIT(code, ier)
```

Description
The function FSUNLAPACKDENSEINIT can be called for Fortran programs to create a LAPACK-based dense SUNLinearSolver object.

Arguments
- `code` (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).

Return value
`ier` is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes
This routine must be called after both the NVVECTOR and SUNMATRIX objects have been initialized.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL_LAPACKDENSE module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.
8.5 The SUNLinearSolver_LapackDense implementation

**FSUNMASSLAPACKDENSEINIT**

Call   
FSUNMASSLAPACKDENSEINIT(ier)

Description The function FSUNMASSLAPACKDENSEINIT can be called for Fortran programs to create a LAPACK-based, dense SUNLinearSolver object for mass matrix linear systems.

Arguments

Return value ier is a int return completion flag equal to 0 for a success return and −1 otherwise. See printed message for details in case of failure.

Notes This routine must be called after both the nvector and sunmatrix mass-matrix objects have been initialized.

8.5.2 SUNLINSOL_LAPACKDENSE description

The SUNLINSOL_LAPACKDENSE module defines the content field of a SUNLinearSolver to be the following structure:

```c
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 (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 (O(N^2) cost).

The SUNLINSOL_LAPACKDENSE module defines dense implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- 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

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 NVVECTOR implementations (NVVECTOR_SERIAL, NVVECTOR_OPENMP, or NVVECTOR_PTHREADS).

8.6.1 SUNLINSOL_LAPACKBAND usage

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 module SUNLINSOL_LAPACKBAND provides the following user-callable routine:

```
SUNLinSolLapackBand
```

Call

\[ LS = \text{SUNLinSolLapackBand}(y, A); \]

Description

The function SUNLinSolLapackBand creates and allocates memory for a LAPACK-based, band SUNLinearSolver object.

Arguments

- \( y \) (N_Vector) a template for cloning vectors needed within the solver
- \( A \) (SUNMatrix) a SUNMATRIX_BAND matrix template for cloning matrices needed within the solver

Return value

This returns a SUNLinearSolver object. If either \( A \) or \( y \) are incompatible then this routine will return NULL.

Notes

This routine will perform consistency checks to ensure that it is called with consistent NVVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX_BAND matrix type and the NVVECTOR_SERIAL, NVVECTOR_OPENMP, and NVVECTOR_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.

For backwards compatibility, we also provide the wrapper functions:

• SUNLapackBand

  Wrapper function for SUNLinSolLapackBand, with identical input and output arguments.

For solvers that include a Fortran interface module, the SUNLINSOL_LAPACKBAND module also includes a Fortran-callable function for creating a SUNLinearSolver object.
8.6 The SUNLinearSolver_LapackBand implementation

**FSUNLAPACKBANDINIT**

Call  
FSUNLAPACKBANDINIT(code, ier)

Description The function FSUNLAPACKBANDINIT can be called for Fortran programs to create a LAPACK-based band SUNLinearSolver object.

Arguments  
code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).

Return value  
ier is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes 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 SUNLINSOL_LAPACKBAND module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

**FSUNMASSLAPACKBANDINIT**

Call  
FSUNMASSLAPACKBANDINIT(ier)

Description The function FSUNMASSLAPACKBANDINIT can be called for Fortran programs to create a LAPACK-based, band SUNLinearSolver object for mass matrix linear systems.

Arguments

Return value  
ier is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called after both the NVECTOR and SUNMATRIX mass-matrix objects have been initialized.

8.6.2 SUNLINSOL_LAPACKBAND description

The SUNLINSOL_LAPACKBAND module defines the content field of a SUNLinearSolver to be the following structure:

```c
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 SUNLINSOL_LAPACKBAND module defines band implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- 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

### 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).

#### 8.7.1 SUNLINSOL_KLU usage

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 module SUNLINSOL_KLU provides the following user-callable routines:

```c
SUNLinSolKLU
Call LS = SUNLinSolKLU(y, A);
Description The function SUNLinSolKLU creates and allocates memory for a SUNLINSOL_KLU object.
Arguments y (N_Vector) a template for cloning vectors needed within the solver
A (SUNMatrix) a SUNMATRIX_SPARSE matrix template for cloning matrices needed within the solver
Return value This returns a SUNLinearSolver object. If either A or y are incompatible then this routine will return NULL.
```
Notes 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. Additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

### SUNLinSol_KLUReInit

**Call**
```
retval = SUNLinSol_KLUReInit(LS, A, nnz, reinit_type);
```

**Description**
The function SUNLinSol_KLUReInit 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 nonzeros has changed or if the structure of the linear system has changed which would require a new symbolic (and numeric factorization).

**Arguments**
- **LS** (SUNLinearSolver) a template for cloning vectors needed within the solver
- **A** (SUNMatrix) a SUNMATRIX_SPARSE matrix template for cloning matrices needed within the solver
- **nnz** (sunindextype) the new number of nonzeros in the matrix
- **reinit_type** (int) flag governing the level of reinitialization. The allowed values are:
  - **SUNKLU_REINIT_FULL** – 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.
  - **SUNKLU_REINIT_PARTIAL** – 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 SUNLinSol_KLUReInit call).

**Return value**
The return values from this function are SUNLS_MEM_NULL (either S or A are NULL), SUNLS_IILL_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.

**Notes**
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. Additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

This routine assumes no other changes to solver use are necessary.

### SUNLinSol_KLUSetOrdering

**Call**
```
retval = SUNLinSol_KLUSetOrdering(LS, ordering);
```

**Description**
This function sets the ordering used by KLU for reducing fill in the linear solve.

**Arguments**
- **LS** (SUNLinearSolver) the SUNLINSOL_KLU object
- **ordering** (int) flag indication the reordering algorithm to use. Options include:
  - 0 AMD,
  - 1 COLAMD, and
  - 2 the natural ordering.
148 Description of the SUNLinearSolver module

The default is 1 for COLAMD.

Return value  The return values from this function are SUNLS_MEM_NULL (S is NULL), SUNLS_ILL_INPUT
(invalid ordering), or SUNLS_SUCCESS.

Notes

For backwards compatibility, we also provide the wrapper functions, each with identical input and
output arguments to the routines that they wrap:

- SUNKLU
  Wrapper function for SUNLinSol_KLU

- SUNKLUREInit
  Wrapper function for SUNLinSol_KLUReInit

- SUNKLUSetOrdering
  Wrapper function for SUNLinSol_KLUSetOrdering

For solvers that include a Fortran interface module, the SUNLINSOL_KLU module also includes a
Fortran-callable function for creating a SUNLinearSolver object.

FSUNKLUINIT

Call    FSUNKLUINIT(code, ier)

Description  The function FSUNKLUINIT can be called for Fortran programs to create a SUNLIN-
SOL_KLU object.

Arguments  code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for
KINSOL, and 4 for ARKODE).

Return value  ier is a return completion flag equal to 0 for a success return and -1 otherwise. See
printed message for details in case of failure.

Notes  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 SUNLINSOL_KLU module in-
cludes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

FSUNMASSKLUINIT

Call    FSUNMASSKLUINIT(ier)

Description  The function FSUNMASSKLUINIT can be called for Fortran programs to create a SUNLIN-
SOL_KLU object for mass matrix linear systems.

Arguments

Return value  ier is an int return completion flag equal to 0 for a success return and -1 otherwise.
See printed message for details in case of failure.

Notes  This routine must be called after both the NVECTOR and SUNMATRIX mass-matrix
objects have been initialized.

The SUNLinSol_KLUReInit and SUNLinSol_KLUSetOrdering routines also support Fortran interfaces
for the system and mass matrix solvers:

FSUNKLUREINIT

Call    FSUNKLUREINIT(code, nnz, reinit_type, ier)

Description  The function FSUNKLUREINIT can be called for Fortran programs to re-initialize a SUN-
LINSOL_KLU object.
8.7 The SUNLinearSolver_KLU implementation

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>(int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).</td>
</tr>
<tr>
<td>nnz</td>
<td>(sunindextype*) the new number of nonzeros in the matrix</td>
</tr>
<tr>
<td>reinit_type</td>
<td>(int*) flag governing the level of reinitialization. The allowed values are:</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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 SUNLinSol_KLUREInit call).</td>
</tr>
</tbody>
</table>

Return value  

ier is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes  

See SUNLinSol_KLUREInit for complete further documentation of this routine.

FSUNMASSKLUREINIT

Call  

FSUNMASSKLUREINIT(nnz, reinit_type, ier)

Description  

The function FSUNMASSKLUREINIT can be called for Fortran programs to re-initialize a SUNLINSOL_KLU object for mass matrix linear systems.

Arguments  

The arguments are identical to FSUNKLUREINIT above, except that code is not needed since mass matrix linear systems only arise in ARKODE.

Return value  

ier is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes  

See SUNLinSol_KLUREInit for complete further documentation of this routine.

FSUNKLUSETORDERING

Call  

FSUNKLUSETORDERING(code, ordering, ier)

Description  

The function FSUNKLUSETORDERING can be called for Fortran programs to change the reordering algorithm used by KLU.

Arguments  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>(int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).</td>
</tr>
<tr>
<td>ordering</td>
<td>(int*) flag indication the reordering algorithm to use. Options include:</td>
</tr>
<tr>
<td></td>
<td>0 AMD,</td>
</tr>
<tr>
<td></td>
<td>1 COLAMD, and</td>
</tr>
<tr>
<td></td>
<td>2 the natural ordering.</td>
</tr>
<tr>
<td></td>
<td>The default is 1 for COLAMD.</td>
</tr>
</tbody>
</table>

Return value  

ier is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes  

See SUNLinSol_KLUSetOrdering for complete further documentation of this routine.

FSUNMASSKLUESTORDERING

Call  

FSUNMASSKLUESTORDERING(ier)

Description  

The function FSUNMASSKLUESTORDERING can be called for Fortran programs to change the reordering algorithm used by KLU for mass matrix linear systems.

Arguments  

The arguments are identical to FSUNKLUSETORDERING above, except that code is not needed since mass matrix linear systems only arise in ARKODE.
Return value \texttt{ier} is a \texttt{int} return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See \texttt{SUNLinSol\_KLU\_SetOrdering} for complete further documentation of this routine.

8.7.2 \texttt{SUNLINSOL\_KLU} description

The \texttt{SUNLINSOL\_KLU} module defines the \textit{content} field of a \texttt{SUNLinearSolver} to be the following structure:

\begin{verbatim}
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*);
};
\end{verbatim}

These entries of the \textit{content} field contain the following information:

\begin{itemize}
  \item \texttt{last_flag} - last error return flag from internal function evaluations,
  \item \texttt{first_factorize} - flag indicating whether the factorization has ever been performed,
  \item \texttt{symbolic} - KLU storage structure for symbolic factorization components,
  \item \texttt{numeric} - KLU storage structure for numeric factorization components,
  \item \texttt{common} - storage structure for common KLU solver components,
  \item \texttt{klu\_solver} - pointer to the appropriate KLU solver function (depending on whether it is using a CSR or CSC sparse matrix).
\end{itemize}

The \texttt{SUNLINSOL\_KLU} module is a SUNLINSOL wrapper for the KLU sparse matrix factorization and solver library written by Tim Davis \cite{1, 9}. In order to use the \texttt{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 \texttt{realtype} set to either \texttt{extended} or \texttt{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 \texttt{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 \texttt{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 \texttt{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 $\varepsilon$ 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 or partial 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 SUNLINSOL_KLU module defines implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- 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

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.

8.8.1 SUNLINSOL_SUPERLUMT usage

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 module SUNLINSOL_SUPERLUMT provides the following user-callable routines:
Description of the SUNLinearSolver module

**SUNLinSol_SuperLUMT**

Call

```c
LS = SUNLinSol_SuperLUMT(y, A, num_threads);
```

Description

The function **SUNLinSol_SuperLUMT** creates and allocates memory for a **SUNLINSOL_SUPERLUMT** object.

Arguments

- `y` (N_Vector) a template for cloning vectors needed within the solver
- `A` (SUNMatrix) a **SUNMATRIX_SPARSE** matrix template for cloning matrices needed within the solver
- `num_threads` (int) desired number of threads (OpenMP or Pthreads, depending on how **SUPERLUMT** was installed) to use during the factorization steps

Return value

This returns a **SUNLinearSolver** object. If either `A` or `y` are incompatible then this routine will return `NULL`.

Notes

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.

The `num_threads` argument is not checked and is passed directly to **SUPERLUMT** routines.

**SUNLinSol_SuperLUMTSetOrdering**

Call

```c
retval = SUNLinSol_SuperLUMTSetOrdering(LS, ordering);
```

Description

The function **SUNLinSol_SuperLUMTSetOrdering** sets the ordering used by **SUPERLUMT** for reducing fill in the linear solve.

Arguments

- `LS` (SUNLinearSolver) the **SUNLINSOL_SUPERLUMT** object
- `ordering` (int) a flag indicating the ordering algorithm, options 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.

Return value

The return values from this function are **SUNLS_MEM_NULL** (S is NULL), **SUNLS_Ill_INPUT** (invalid `ordering_choice`), or **SUNLS_SUCCESS**.

Notes

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

- **SUNSUPERLUMT**
  Wrapper function for **SUNLinSol_SuperLUMT**

- **SUNSUPERLUMTSetOrdering**
  Wrapper function for **SUNLinSol_SuperLUMTSetOrdering**

For solvers that include a Fortran interface module, the **SUNLINSOL_SUPERLUMT** module also includes a Fortran-callable function for creating a **SUNLinearSolver** object.
The function `FSUNSUPERLUMTINIT` can be called for Fortran programs to create a SUNLINSOL_KLU object. The function `FSUNMASSSUPERLUMTINIT` can be called for Fortran programs to create a SUNLINSOL_SUPERLUMT object for mass matrix linear systems. The function `FSUNSUPERLUMTSETORDERING` can be called for Fortran programs to update the ordering algorithm in a SUNLINSOL_SUPERLUMT object.
**FSUNMASSUPERLUMTSETORDERING**

**Call**

`FSUNMASSUPERLUMTSETORDERING(ordering, ier)`

**Description**
The function `FSUNMASSUPERLUMTSETORDERING` can be called for Fortran programs to update the ordering algorithm in a `SUNLINSOL_SUPERLUMT` object for mass matrix linear systems.

**Arguments**
- `ordering` (int*) a flag indicating the ordering algorithm, options 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.

**Return value**
- `ier` is an int return completion flag equal to 0 for a success return and ~1 otherwise. See printed message for details in case of failure.

**Notes**
- See `SUNLinSol_SuperLUMTSetOrdering` for complete further documentation of this routine.

### 8.8.2 SUNLINSOL_SUPERLUMT description

The `SUNLINSOL_SUPERLUMT` module defines the `content` field of a `SUNLinearSolver` to be the following structure:

```c
struct _SUNLinearSolverContent_SuperLUMT {
    long int last_flag;
    int first_factorize;
    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.
8.9 The SUNLinearSolver_SPGMR implementation

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 SUNLINSOL_SUPERLUMT module defines implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- 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

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 ($N\_VC\_clone, N\_VDotProd, N\_VScale, N\_VLinearSum, N\_VProd, N\_VConst, N\_VDiv,$ and $N\_VDestroy$). When using Classical Gram-Schmidt, the optional function $N\_VDotProdMulti$ may be supplied for increased efficiency.
8.9.1 SUNLINSOL_SPGMR usage

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 module SUNLINSOL_SPGMR provides the following user-callable routines:

\begin{verbatim}
Call LS = SUNLinSol_SPGMR(y, pretype, maxl);
Description The function SUNLinSol_SPGMR creates and allocates memory for a SPGMR SUNLinearSolver.
Arguments y (N_Vector) a template for cloning vectors needed within the solver
pretype (int) flag indicating the desired type of preconditioning, allowed values are:
  - PREC_NONE (0)
  - PREC_LEFT (1)
  - PREC_RIGHT (2)
  - PREC_BOTH (3)
maxl (int) the number of Krylov basis vectors to use. values ≤ 0 will result in the default value (5).
Return value This returns a SUNLinearSolver object. If either y is incompatible then this routine will return NULL.
Notes 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.
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.
\end{verbatim}

\begin{verbatim}
Call retval = SUNLinSol_SPGMRSetPrecType(LS, pretype);
Description The function SUNLinSol_SPGMRSetPrecType updates the type of preconditioning to use in the SUNLINSOL_SPGMR object.
Arguments LS (SUNLinearSolver) the SUNLINSOL_SPGMR object to update
pretype (int) flag indicating the desired type of preconditioning, allowed values match those discussed in SUNLinSol_SPGMR.
Return value This routine will return with one of the error codes SUNLS_Ill_INPUT (illegal pretype), SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.
Notes
\end{verbatim}

\begin{verbatim}
Call retval = SUNLinSol_SPGMRSetGSType(LS, gstype);
Description The function SUNLinSol_SPGMRSetGSType sets the type of Gram-Schmidt orthogonalization to use in the SUNLINSOL_SPGMR object.
Arguments LS (SUNLinearSolver) the SUNLINSOL_SPGMR object to update
gstype (int) flag indicating the desired orthogonalization algorithm; allowed values are:
\end{verbatim}
8.9 The SUNLinearSolver_SPGMR implementation

- MODIFIED_GS (1)
- CLASSICAL_GS (2)

Any other integer input will result in a failure, returning error code SUNLS_Ill_INPUT.

Return value This routine will return with one of the error codes SUNLS_Ill_INPUT (illegal pretype), SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

Notes

SUNLinSol_SPGMRSetMaxRestarts

Call retval = SUNLinSol_SPGMRSetMaxRestarts(LS, maxrs);

Description The function SUNLinSol_SPGMRSetMaxRestarts sets the number of GMRES restarts to allow in the SUNLINSOL_SPGMR object.

Arguments LS (SUNLinearSolver) the SUNLINSOL_SPGMR object to update
maxrs (int) integer indicating number of restarts to allow. A negative input will result in the default of 0.

Return value This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

Notes

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

- SUNSPGMR
  Wrapper function for SUNLinSol_SPGMR
- SUNSPGMRSetPrecType
  Wrapper function for SUNLinSol_SPGMRSetPrecType
- SUNSPGMRSetGSType
  Wrapper function for SUNLinSol_SPGMRSetGSType
- SUNSPGMRSetMaxRestarts
  Wrapper function for SUNLinSol_SPGMRSetMaxRestarts

For solvers that include a Fortran interface module, the SUNLINSOL_SPGMR module also includes a Fortran-callable function for creating a SUNLinearSolver object.

FSUNSPGMRINIT

Call FSUNSPGMRINIT(code, pretype, maxl, ier)

Description The function FSUNSPGMRINIT can be called for Fortran programs to create a SUNLINSOL_SPGMR object.

Arguments code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
pretype (int*) flag indicating desired preconditioning type
maxl (int*) flag indicating Krylov subspace size

Return value ier is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called after the NVECTOR object has been initialized.

Allowable values for pretype and maxl are the same as for the C function SUNLinSol_SPGMR.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL_SPGMR module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.
FSUNMASSSPGMRINIT
Call FSUNMASSSPGMRINIT(pretype, maxl, ier)
Description The function FSUNMASSSPGMRINIT can be called for Fortran programs to create a SUN- LINSOL_SPGMR object for mass matrix linear systems.
Arguments
pretype (int*) flag indicating desired preconditioning type
maxl (int*) flag indicating Krylov subspace size
Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes This routine must be called after the NVECTOR object has been initialized.
Allowable values for pretype and maxl are the same as for the C function SUNLinSol_SPGMR.
The SUNLinSol_SPGMRSetPrecType, SUNLinSol_SPGMRSetGSType and SUNLinSol_SPGMRSetMaxRestarts routines also support Fortran interfaces for the system and mass matrix solvers.

FSUNSPGMRSETGSTYPE
Call FSUNSPGMRSETGSTYPE(code, gstype, ier)
Description The function FSUNSPGMRSETGSTYPE can be called for Fortran programs to change the Gram-Schmidt orthogonalization algorithm.
Arguments
code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
gstype (int*) flag indicating the desired orthogonalization algorithm.
Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes See SUNLinSol_SPGMRSets for complete further documentation of this routine.

FSUNMASSSPGMRSETGSTYPE
Call FSUNMASSSPGMRSETGSTYPE(gstype, ier)
Description The function FSUNMASSSPGMRSETGSTYPE can be called for Fortran programs to change the Gram-Schmidt orthogonalization algorithm for mass matrix linear systems.
Arguments The arguments are identical to FSUNSPGMRSETGSTYPE above, except that code is not needed since mass matrix linear systems only arise in ARKODE.
Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes See SUNLinSol_SPGMRSetGSType for complete further documentation of this routine.

FSUNSPGMRSETPRECTYPE
Call FSUNSPGMRSETPRECTYPE(code, pretype, ier)
Description The function FSUNSPGMRSETPRECTYPE can be called for Fortran programs to change the type of preconditioning to use.
Arguments
code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
pretype (int*) flag indicating the type of preconditioning to use.
Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes See SUNLinSol_SPGMRSetPrecType for complete further documentation of this routine.
The SUNLinearSolver_SPGMR implementation

**FSUNMASSSPGMRSETPRECTYPE**

Call: `FSUNMASSSPGMRSETPRECTYPE(pretype, ier)`

Description: The function `FSUNMASSSPGMRSETPRECTYPE` can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

Arguments: The arguments are identical to `FSUNSPGMRSETPRECTYPE` above, except that `code` is not needed since mass matrix linear systems only arise in ARKODE.

Return value: `ier` is an `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes: See `SUNLinSol_SPGMRSetPrecType` for complete further documentation of this routine.

**FSUNSPGMRSETMAXRS**

Call: `FSUNSPGMRSETMAXRS(code, maxrs, ier)`

Description: The function `FSUNSPGMRSETMAXRS` can be called for Fortran programs to change the maximum number of restarts allowed for SPGMR.

Arguments: `code` (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).

`maxrs` (int*) maximum allowed number of restarts.

Return value: `ier` is an `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes: See `SUNLinSol_SPGMRSetMaxRestarts` for complete further documentation of this routine.

**FSUNMASSSPGMRSETMAXRS**

Call: `FSUNMASSSPGMRSETMAXRS(maxrs, ier)`

Description: The function `FSUNMASSSPGMRSETMAXRS` can be called for Fortran programs to change the maximum number of restarts allowed for SPGMR for mass matrix linear systems.

Arguments: The arguments are identical to `FSUNSPGMRSETMAXRS` above, except that `code` is not needed since mass matrix linear systems only arise in ARKODE.

Return value: `ier` is an `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes: See `SUNLinSol_SPGMRSetMaxRestarts` for complete further documentation of this routine.

### 8.9.2 SUNLINSOL_SPGMR description

The `SUNLINSOL_SPGMR` module defines the `content` field of a `SUNLinearSolver` to be the following structure:

```c
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),
umiters - 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 v1, . . . , v_{maxl+1}, stored in V[0], . . . , V[maxl]. Each v_i is a vector of type NVECTOR.,
Hes - the (maxl + 1) × 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, . . . , 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 SUNLINSOL_SPGMR module defines implementations of all “iterative” linear solver operations listed in Sections 8.0.1-8.0.3:

- 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

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). When using Classical Gram-Schmidt, the optional function N_VDotProdMulti may be supplied for increased efficiency. 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).
8.10.1 SUNLINSOL_SPFGMR usage

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_sunlinsol_spfgmr` module library.

The module SUNLINSOL_SPFGMR provides the following user-callable routines:

### SUNLinSol_SPFGMR

**Call**

```c
LS = SUNLinSol_SPFGMR(y, pretype, maxl);
```

**Description**

The function `SUNLinSol_SPFGMR` creates and allocates memory for a SPFGMR SUNLinearSolver.

**Arguments**

- `y` (`N_Vector`) a template for cloning vectors needed within the solver
- `pretype` (int) flag indicating the desired type of preconditioning, allowed values are:
  - PREC_NONE (0)
  - PREC_LEFT (1)
  - PREC_RIGHT (2)
  - PREC_BOTH (3)
  
  Any other integer input will result in the default (no preconditioning).
- `maxl` (int) the number of Krylov basis vectors to use. values $\leq 0$ will result in the default value (5).

**Return value**

This returns a SUNLinearSolver object. If either `y` is incompatible then this routine will return NULL.

**Notes**

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.

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.

### SUNLinSol_SPFGMRSetPrecType

**Call**

```c
retval = SUNLinSol_SPFGMRSetPrecType(LS, pretype);
```

**Description**

The function `SUNLinSol_SPFGMRSetPrecType` updates the type of preconditioning to use in the SUNLINSOL_SPFGMR object.

**Arguments**

- `LS` (SUNLinearSolver) the SUNLINSOL_SPFGMR object to update
- `pretype` (int) flag indicating the desired type of preconditioning, allowed values match those discussed in `SUNLinSol_SPFGMR`.

**Return value**

This routine will return with one of the error codes `SUNLS_Ill_INPUT` (illegal `pretype`), `SUNLS_MEM_NULL` (`S` is NULL) or `SUNLS_SUCCESS`.

**Notes**

### SUNLinSol_SPFGMRSetGSType

**Call**

```c
retval = SUNLinSol_SPFGMRSetGSType(LS, gstype);
```

**Description**

The function `SUNLinSol_SPFGMRSetGSType` sets the type of Gram-Schmidt orthogonalization to use in the SUNLINSOL_SPFGMR object.

**Arguments**

- `LS` (SUNLinearSolver) the SUNLINSOL_SPFGMR object to update
- `gstype` (int) flag indicating the desired orthogonalization algorithm; allowed values are:
• MODIFIED_GS (1)
• CLASSICAL_GS (2)

Any other integer input will result in a failure, returning error code SUNLS_INPUT.

Return value This routine will return with one of the error codes SUNLS_INPUT (illegal pretype), SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

Notes

[SUNLinSol_SPFGMRSetMaxRestarts]

Call retval = SUNLinSol_SPFGMRSetMaxRestarts(LS, maxrs);

Description The function SUNLinSol_SPFGMRSetMaxRestarts sets the number of GMRES restarts to allow in the SUNLINSOL_SPFGMR object.

Arguments LS (SUNLinearSolver) the SUNLINSOL_SPFGMR object to update
maxrs (int) integer indicating number of restarts to allow. A negative input will result in the default of 0.

Return value This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

Notes

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

• SUNSPFGMR
  Wrapper function for SUNLinSol_SPFGMR

• SUNSPFGMRSetPrecType
  Wrapper function for SUNLinSol_SPFGMRSetPrecType

• SUNSPFGMRSetGSType
  Wrapper function for SUNLinSol_SPFGMRSetGSType

• SUNSPFGMRSetMaxRestarts
  Wrapper function for SUNLinSol_SPFGMRSetMaxRestarts

For solvers that include a Fortran interface module, the SUNLINSOL_SPFGMR module also includes a Fortran-callable function for creating a SUNLinearSolver object.

[FSUNSPFGMRINIT]

Call FSUNSPFGMRINIT(code, pretype, maxl, ier)

Description The function FSUNSPFGMRINIT can be called for Fortran programs to create a SUNLINSOL_SPFGMR object.

Arguments code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
pretype (int*) flag indicating desired preconditioning type
maxl (int*) flag indicating Krylov subspace size

Return value ier is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called after the NVECTOR object has been initialized.

Allowable values for pretype and maxl are the same as for the C function SUNLinSol_SPFGMR.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL_SPFGMR module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.
FSUNMASSSPFGMRINIT
Call FSUNMASSSPFGMRINIT(pretype, maxl, ier)
Description The function FSUNMASSSPFGMRINIT can be called for Fortran programs to create a SUN-LINSOL_SPFGMR object for mass matrix linear systems.
Arguments pretype (int*) flag indicating desired preconditioning type
maxl (int*) flag indicating Krylov subspace size
Return value ier is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes This routine must be called after the NVECTOR object has been initialized.
Allowable values for pretype and maxl are the same as for the C function SUNLinSol_SPFGMR.
The SUNLinSol_SPFGMRSetPrecType, SUNLinSol_SPFGMRSetGSType and SUNLinSol_SPFGMRSetMaxRestarts routines also support Fortran interfaces for the system and mass matrix solvers.

FSUNSPFGMRSETGSTYPE
Call FSUNSPFGMRSETGSTYPE(code, gstype, ier)
Description The function FSUNSPFGMRSETGSTYPE can be called for Fortran programs to change the Gram-Schmidt orthogonalization algorithm.
Arguments code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
gstype (int*) flag indicating the desired orthogonalization algorithm.
Return value ier is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes See SUNLinSol_SPFGMRSetGSType for complete further documentation of this routine.

FSUNMASSSPFGMRSETGSTYPE
Call FSUNMASSSPFGMRSETGSTYPE(gstype, ier)
Description The function FSUNMASSSPFGMRSETGSTYPE can be called for Fortran programs to change the Gram-Schmidt orthogonalization algorithm for mass matrix linear systems.
Arguments The arguments are identical to FSUNSPFGMRSETGSTYPE above, except that code is not needed since mass matrix linear systems only arise in ARKODE.
Return value ier is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes See SUNLinSol_SPFGMRSetGSType for complete further documentation of this routine.

FSUNSPFGMRSETPRECYPE
Call FSUNSPFGMRSETPRECYPE(code, pretype, ier)
Description The function FSUNSPFGMRSETPRECYPE can be called for Fortran programs to change the type of preconditioning to use.
Arguments code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
pretype (int*) flag indication the type of preconditioning to use.
Return value ier is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes See SUNLinSol_SPFGMRSetPrecType for complete further documentation of this routine.
8.10 The SUNLinearSolver_SPFGMR implementation

**FSUNMASSSPFGMRSETPRECTYPE**

Call

FSUNMASSSPFGMRSETPRECTYPE(pretype, ier)

Description

The function FSUNMASSSPFGMRSETPRECTYPE can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

Arguments

The arguments are identical to FSUNSPFGMRSETPRECTYPE above, except that code is not needed since mass matrix linear systems only arise in ARKODE.

Return value

ier is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes

See SUNLinSol_SPFGMRSetPrecType for complete further documentation of this routine.

**FSUNSPFGMRSETMAXRS**

Call

FSUNSPFGMRSETMAXRS(code, maxrs, ier)

Description

The function FSUNSPFGMRSETMAXRS can be called for Fortran programs to change the maximum number of restarts allowed for SPFGMR.

Arguments

code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSL, and 4 for ARKODE).

maxrs (int*) maximum allowed number of restarts.

Return value

ier is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes

See SUNLinSol_SPFGMRSetMaxRestarts for complete further documentation of this routine.

**FSUNMASSSPFGMRSETMAXRS**

Call

FSUNMASSSPFGMRSETMAXRS(maxrs, ier)

Description

The function FSUNMASSSPFGMRSETMAXRS can be called for Fortran programs to change the maximum number of restarts allowed for SPFGMR for mass matrix linear systems.

Arguments

The arguments are identical to FSUNSPFGMRSETMAXRS above, except that code is not needed since mass matrix linear systems only arise in ARKODE.

Return value

ier is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes

See SUNLinSol_SPFGMRSetMaxRestarts for complete further documentation of this routine.

8.10.2 SUNLINSOL_SPFGMR description

The SUNLINSOL_SPFGMR module defines the content field of a SUNLinearSolver to be the following structure:

```c
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),
umiters - 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 \(v_1, \ldots, v_{\text{maxl}+1}\), stored in \(V[0], \ldots, V[\text{maxl}]\). Each \(v_i\) is a vector of type nvector.,
Z - the array of preconditioned Krylov basis vectors \(z_1, \ldots, z_{\text{maxl}+1}\), stored in \(Z[0], \ldots, Z[\text{maxl}]\). Each \(z_i\) is a vector of type nvector.,
Hes - the \((\text{maxl} + 1) \times \text{maxl}\) Hessenberg matrix. It is stored row-wise so that the \((i,j)\)th element is given by \(Hes[i][j]\),
givens - a length \(2\times\text{maxl}\) array which represents the Givens rotation matrices that arise in the FGMRES algorithm. These matrices are \(F_0, F_1, \ldots, F_j\), where \(F_i = \begin{bmatrix}
1 & & & \\
& 1 & & \\
& & c_i - s_i & \\
& & s_i & c_i \\
& & & 1 \\
& & & & & & & 1
\end{bmatrix}\), are represented in the givens vector as \(\text{givens}[0] = c_0, \text{givens}[1] = s_0, \text{givens}[2] = c_1, \text{givens}[3] = s_1, \ldots, \text{givens}[2j] = c_j, \text{givens}[2j+1] = s_j,\)
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),

vttemp - 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 SUNLINSOL_SPFGMR module defines implementations of all “iterative” linear solver operations listed in Sections 8.0.1-8.0.3:

• 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

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.
8.11.1 SUNLINSOL_SPBCGS usage

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 lib sundials_sunlinsolspbcgs module library.

The module SUNLINSOL_SPBCGS provides the following user-callable routines:

**SUNLinSol_SPBCGS**

**Call**

\[
\text{LS} = \text{SUNLinSol_SPBCGS}(y, \text{pretype}, \text{maxl}); 
\]

**Description**
The function SUNLinSol_SPBCGS creates and allocates memory for a SPBCGS SUNLinearSolver.

**Arguments**
- \(y\) (N_Vector) a template for cloning vectors needed within the solver
- \(\text{pretype}\) (int) flag indicating the desired type of preconditioning, allowed values are:
  - \(\text{PREC_NONE}\) (0)
  - \(\text{PREC_LEFT}\) (1)
  - \(\text{PREC_RIGHT}\) (2)
  - \(\text{PREC_BOTH}\) (3)

Any other integer input will result in the default (no preconditioning).
- \(\text{maxl}\) (int) the number of linear iterations to allow; values \(\leq 0\) will result in the default value (5).

**Return value**
This returns a SUNLinearSolver object. If either \(y\) is incompatible then this routine will return NULL.

**Notes**
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.

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.

**SUNLinSol_SPBCGSSetPrecType**

**Call**

\[
\text{retval} = \text{SUNLinSol_SPBCGSSetPrecType}(\text{LS}, \text{pretype}); 
\]

**Description**
The function SUNLinSol_SPBCGSSetPrecType updates the type of preconditioning to use in the SUNLINSOL_SPBCGS object.

**Arguments**
- \(\text{LS}\) (SUNLinearSolver) the SUNLINSOL_SPBCGS object to update
- \(\text{pretype}\) (int) flag indicating the desired type of preconditioning, allowed values match those discussed in SUNLinSol_SPBCGS.

**Return value**
This routine will return with one of the error codes SUNLS_ILL_INPUT (illegal \(\text{pretype}\)), SUNLS_MEM_NULL (\(S\) is NULL) or SUNLS_SUCCESS.

**Notes**

**SUNLinSol_SPBCGSSetMaxl**

**Call**

\[
\text{retval} = \text{SUNLinSol_SPBCGSSetMaxl}(\text{LS}, \text{maxl}); 
\]

**Description**
The function SUNLinSol_SPBCGSSetMaxl updates the number of linear solver iterations to allow.

**Arguments**
- \(\text{LS}\) (SUNLinearSolver) the SUNLINSOL_SPBCGS object to update
- \(\text{maxl}\) (int) flag indicating the number of iterations to allow; values \(\leq 0\) will result in the default value (5).
Return value
This routine will return with one of the error codes `SUNLS_MEM_NULL` (S is NULL) or `SUNLS_SUCCESS`.

Notes
For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

- **SUNSPBCGS**
  
  Wrapper function for `SUNLinSol_SPBCGS`

- **SUNSPBCGSSetPrecType**
  
  Wrapper function for `SUNLinSol_SPBCGSSetPrecType`

- **SUNSPBCGSSetMaxl**
  
  Wrapper function for `SUNLinSol_SPBCGSSetMaxl`

For solvers that include a Fortran interface module, the `SUNLinSol_SPBCGS` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

```fortran
FSUNSPBCGSINIT
Call FSUNSPBCGSINIT(code, pretype, maxl, ier)
Description The function FSUNSPBCGSINIT can be called for Fortran programs to create a SUNLIN-
SOL_SPBCGS object.
Arguments code  (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 
for KINSOL, and 4 for ARKODE).
pretype (int*) flag indicating desired preconditioning type
maxl (int*) flag indicating number of iterations to allow
Return value ier is a return completion flag equal to 0 for a success return and -1 otherwise. See 
printed message for details in case of failure.
Notes This routine must be called after the NVECTOR object has been initialized.
Allowable values for pretype and maxl are the same as for the C function SUNLinSol_SPBCGS.
Additionally, when using ARKODE with a non-identity mass matrix, the SUNLIN_SOL_SPBCGS module 
includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.
```

```fortran
FSUNMASSSPBCGSINIT
Call FSUNMASSSPBCGSINIT(pretype, maxl, ier)
Description The function FSUNMASSSPBCGSINIT can be called for Fortran programs to create a SUN-
LINSOL_SPBCGS object for mass matrix linear systems.
Arguments pretype (int*) flag indicating desired preconditioning type
maxl (int*) flag indicating number of iterations to allow
Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise. 
See printed message for details in case of failure.
Notes This routine must be called after the NVECTOR object has been initialized.
Allowable values for pretype and maxl are the same as for the C function SUNLinSol_SPBCGS.
```

The `SUNLinSol_SPBCGSSetPrecType` and `SUNLinSol_SPBCGSSetMaxl` routines also support Fortran 
interfaces for the system and mass matrix solvers.
**FSUNSPBCGSSETPRECTYPE**

**Call**

FSUNSPBCGSSETPRECTYPE(code, pretype, ier)

**Description**
The function `FSUNSPBCGSSETPRECTYPE` can be called for Fortran programs to change the type of preconditioning to use.

**Arguments**
- `code` (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
- `pretype` (int*) flag indication the type of preconditioning to use.

**Return value**
`ier` is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes**
See `SUNLinSol_SPBCGSSetPrecType` for complete further documentation of this routine.

---

**FSUNMASSSPBCGSSETPRECTYPE**

**Call**

FSUNMASSSPBCGSSETPRECTYPE(pretype, ier)

**Description**
The function `FSUNMASSSPBCGSSETPRECTYPE` can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

**Arguments**
The arguments are identical to `FSUNSPBCGSSETPRECTYPE` above, except that `code` is not needed since mass matrix linear systems only arise in ARKODE.

**Return value**
`ier` is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes**
See `SUNLinSol_SPBCGSSetPrecType` for complete further documentation of this routine.

---

**FSUNSPBCGSSETMAXL**

**Call**

FSUNSPBCGSSETMAXL(code, maxl, ier)

**Description**
The function `FSUNSPBCGSSETMAXL` can be called for Fortran programs to change the maximum number of iterations to allow.

**Arguments**
- `code` (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
- `maxl` (int*) the number of iterations to allow.

**Return value**
`ier` is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes**
See `SUNLinSol_SPBCGSSetMaxl` for complete further documentation of this routine.

---

**FSUNMASSSPBCGSSETMAXL**

**Call**

FSUNMASSSPBCGSSETMAXL(maxl, ier)

**Description**
The function `FSUNMASSSPBCGSSETMAXL` can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

**Arguments**
The arguments are identical to `FSUNSPBCGSSETMAXL` above, except that `code` is not needed since mass matrix linear systems only arise in ARKODE.

**Return value**
`ier` is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes**
See `SUNLinSol_SPBCGSSetMaxl` for complete further documentation of this routine.
8.11.2 SUNLINSOL_SPBCGS description

The SUNLINSOL_SPBCGS module defines the content field of a SUNLinearSolver to be the following structure:

```c
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 SUNLINSOL_SPBCGS module defines implementations of all “iterative” linear solver operations listed in Sections 8.0.1-8.0.3:

- 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

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_VDot Prod, 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.

8.12.1 SUNLINSOL_SPTFQMR usage

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 module SUNLINSOL_SPTFQMR provides the following user-callable routines:
8.12 The SUNLinearSolver_SPTFQMR implementation

**SUNLinSol_SPTFQMR**

**Call**

\[
LS = \text{SUNLinSol\_SPTFQMR}(y, \text{pretype}, \text{maxl});
\]

**Description**

The function SUNLinSol_SPTFQMR creates and allocates memory for a SPTFQMR SUNLinearSolver.

**Arguments**

- \(y\) (N\_Vector) a template for cloning vectors needed within the solver
- \(\text{pretype}\) (int) flag indicating the desired type of preconditioning, allowed values are:
  - PREC\_NONE (0)
  - PREC\_LEFT (1)
  - PREC\_RIGHT (2)
  - PREC\_BOTH (3)

  Any other integer input will result in the default (no preconditioning).
- \(\text{maxl}\) (int) the number of linear iterations to allow; values \(\leq 0\) will result in the default value (5).

**Return value**

This returns a SUNLinearSolver object. If either \(y\) is incompatible then this routine will return NULL.

**Notes**

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.

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.

**SUNLinSol_SPTFQMRSetPrecType**

**Call**

\[
\text{retval} = \text{SUNLinSol\_SPTFQMRSetPrecType}(\text{LS}, \text{pretype});
\]

**Description**

The function SUNLinSol_SPTFQMRSetPrecType updates the type of preconditioning to use in the SUNLINSOL_SPTFQMR object.

**Arguments**

- \(\text{LS}\) (SUNLinearSolver) the SUNLINSOL_SPTFQMR object to update
- \(\text{pretype}\) (int) flag indicating the desired type of preconditioning, allowed values match those discussed in SUNLinSol_SPTFQMR.

**Return value**

This routine will return with one of the error codes SUNLS\_ILL\_INPUT (illegal \(\text{pretype}\)), SUNLS\_MEM\_NULL (S is NULL) or SUNLS\_SUCCESS.

**Notes**

**SUNLinSol_SPTFQMRSetMaxl**

**Call**

\[
\text{retval} = \text{SUNLinSol\_SPTFQMRSetMaxl}(\text{LS}, \text{maxl});
\]

**Description**

The function SUNLinSol_SPTFQMRSetMaxl updates the number of linear solver iterations to allow.

**Arguments**

- \(\text{LS}\) (SUNLinearSolver) the SUNLINSOL_SPTFQMR object to update
- \(\text{maxl}\) (int) flag indicating the number of iterations to allow; values \(\leq 0\) will result in the default value (5)

**Return value**

This routine will return with one of the error codes SUNLS\_MEM\_NULL (S is NULL) or SUNLS\_SUCCESS.

**Notes**

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:
• **SUNSPTFQMR**
  Wrapper function for SUNLinSol SPTFQMR

• **SUNSPTFQMRSetPrecType**
  Wrapper function for SUNLinSol SPTFQMRSetPrecType

• **SUNSPTFQMRSetMaxl**
  Wrapper function for SUNLinSol SPTFQMRSetMaxl

For solvers that include a Fortran interface module, the SUNLIN_SOL_SPTFQMR module also includes a Fortran-callable function for creating a SUNLinearSolver object.

### FSUNSPTFQMRINIT
Call FSUNSPTFQMRINIT(code, pretype, maxl, ier)

**Description** The function FSUNSPTFQMRINIT can be called for Fortran programs to create a SUNLIN_SOL_SPTFQMR object.

**Arguments**
- `code` (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
- `pretype` (int*) flag indicating desired preconditioning type
- `maxl` (int*) flag indicating number of iterations to allow

**Return value** `ier` is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes** This routine must be called after the NVECTOR object has been initialized.

Allowable values for `pretype` and `maxl` are the same as for the C function SUNLinSol SPTFQMR.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLIN_SOL_SPTFQMR module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

### FSUNMASSSPTFQMRINIT
Call FSUNMASSSPTFQMRINIT(pretype, maxl, ier)

**Description** The function FSUNMASSSPTFQMRINIT can be called for Fortran programs to create a SUNLIN_SOL_SPTFQMR object for mass matrix linear systems.

**Arguments**
- `pretype` (int*) flag indicating desired preconditioning type
- `maxl` (int*) flag indicating number of iterations to allow

**Return value** `ier` is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

**Notes** This routine must be called after the NVECTOR object has been initialized.

Allowable values for `pretype` and `maxl` are the same as for the C function SUNLinSol SPTFQMR.

The SUNLinSol SPTFQMRSetPrecType and SUNLinSol SPTFQMRSetMaxl routines also support Fortran interfaces for the system and mass matrix solvers.

### FSUNSPTFQMRSETPRECTYPE
Call FSUNSPTFQMRSETPRECTYPE(code, pretype, ier)

**Description** The function FSUNSPTFQMRSETPRECTYPE can be called for Fortran programs to change the type of preconditioning to use.

**Arguments**
- `code` (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
- `pretype` (int*) flag indication the type of preconditioning to use.
8.12 The SUNLinearSolver_SPTFQMR implementation

Return value \( \text{ier} \) is an \texttt{int} return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol_SPTFQMRSetPrecType for complete further documentation of this routine.

\textbf{FSUNMASSSPTFQMRSETPRECTYPE}

Call \texttt{FSUNMASSSPTFQMRSETPRECTYPE(pretype, ier)}

Description The function \texttt{FSUNMASSSPTFQMRSETPRECTYPE} can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to \texttt{FSUNSPTFQMRSETPRECTYPE} above, except that \texttt{code} is not needed since mass matrix linear systems only arise in ARKODE.

Return value \( \text{ier} \) is an \texttt{int} return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol_SPTFQMRSetPrecType for complete further documentation of this routine.

\textbf{FSUNSPTFQMRSETMAXL}

Call \texttt{FSUNSPTFQMRSETMAXL(code, maxl, ier)}

Description The function \texttt{FSUNSPTFQMRSETMAXL} can be called for Fortran programs to change the maximum number of iterations to allow.

Arguments \texttt{code (int*)} is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).

\texttt{maxl (int*)} the number of iterations to allow

Return value \( \text{ier} \) is an \texttt{int} return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol_SPTFQMRSetMaxl for complete further documentation of this routine.

\textbf{FSUNMASSSPTFQMRSETMAXL}

Call \texttt{FSUNMASSSPTFQMRSETMAXL(maxl, ier)}

Description The function \texttt{FSUNMASSSPTFQMRSETMAXL} can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to \texttt{FSUNSPTFQMRSETMAXL} above, except that \texttt{code} is not needed since mass matrix linear systems only arise in ARKODE.

Return value \( \text{ier} \) is an \texttt{int} return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol_SPTFQMRSetMaxl for complete further documentation of this routine.

8.12.2 SUNLINSOL_SPTFQMR description

The SUNLINSOL_SPTFQMR module defines the \texttt{content} field of a SUNLinearSolver to be the following structure:

\begin{verbatim}
struct _SUNLinearSolverContent_SPTFQMR {
    int maxl;
    int pretype;
    int numiters;
    realtype resnorm;
    long int last_flag;
}
\end{verbatim}
ATimesFn ATimes;
void* ATData;
PSetupFn Psetup;
P SolveFn 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),
umiters - 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.
8.13 The SUNLinearSolver_PCW implementation

The SUNLinearSolver_PCW implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_PCW, 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, PCW 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, PCW 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 PCW 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
\[
\tilde{A}\tilde{x} = \tilde{b}
\] (8.3)
where
\[
\tilde{A} = SP^{-1}AP^{-1}S,
\] 
\[
\tilde{b} = SP^{-1}b,
\] 
\[
\tilde{x} = S^{-1}P_{x}.
\] (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:
\[
\|\tilde{b} - \tilde{A}\tilde{x}\|_2 < \delta
\]
\[
\Leftrightarrow
\]
\[
\|SP^{-1}b - SP^{-1}Ax\|_2 < \delta
\]
\[
\Leftrightarrow
\]
\[
\|P^{-1}b - P^{-1}Ax\|_S < \delta
\]
where $\|v\|_S = \sqrt{v^T S^T S v}$, with an input tolerance $\delta$.

### 8.13.1 SUNLINSOL_PCG usage

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 module SUNLINSOL_PCG provides the following user-callable routines:

```
SUNLinSol_PCG
Call LS = SUNLinSol_PCG(y, pretype, maxl);
Description The function SUNLinSol_PCG creates and allocates memory for a PCG SUNLinearSolver.
Arguments y (N_Vector) a template for cloning vectors needed within the solver
pretype (int) flag indicating whether to use preconditioning. 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).
maxl (int) the number of linear iterations to allow; values $\leq 0$ will result in the default value (5).
Return value This returns a SUNLinearSolver object. If either $y$ is incompatible then this routine will return NULL.
Notes 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.
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.
```
8.13 The SUNLinearSolver_PCG implementation

**SUNLinSol_PCGSetPrecType**

Call: `retval = SUNLinSol_PCGSetPrecType(LS, pretype);`

Description: The function `SUNLinSol_PCGSetPrecType` updates the flag indicating use of preconditioning in the `SUNLinSol_PCG` object.

Arguments:
- `LS` (SUNLinearSolver) the `SUNLinSol_PCG` object to update
- `pretype` (int) flag indicating use of preconditioning. Allowable values match those discussed in `SUNLinSol_PCG`.

Return value: This routine will return with one of the error codes `SUNLS_Ill_INPUT` (illegal `pretype`), `SUNLS_MEM_NULL` (S is NULL) or `SUNLS_SUCCESS`.

Notes: For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:
- **SUNPCG**
  - Wrapper function for `SUNLinSol_PCG`
- **SUNPCGSetPrecType**
  - Wrapper function for `SUNLinSol_PCGSetPrecType`
- **SUNPCGSetMaxl**
  - Wrapper function for `SUNLinSol_PCGSetMaxl`

For solvers that include a Fortran interface module, the `SUNLinSol_PCG` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

**FSUNPCGINIT**

Call: `FSUNPCGINIT(code, pretype, maxl, ier)`

Description: The function `FSUNPCGINIT` can be called for Fortran programs to create a `SUNLinSol_PCG` object.

Arguments:
- `code` (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
- `pretype` (int*) flag indicating desired preconditioning type
- `maxl` (int*) flag indicating number of iterations to allow

Return value: `ier` is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes: This routine must be called after the `NVECTOR` object has been initialized.

Allowable values for `pretype` and `maxl` are the same as for the C function `SUNLinSol_PCG`.

Additionally, when using ARKODE with a non-identity mass matrix, the `SUNLinSol_PCG` module includes a Fortran-callable function for creating a `SUNLinearSolver` mass matrix solver object.
FSUNMASSPCGINIT

call FSUNMASSPCGINIT(pretype, maxl, ier)

description The function FSUNMASSPCGINIT can be called for Fortran programs to create a SUNLINSOL_PCG object for mass matrix linear systems.

arguments pretype (int*) flag indicating desired preconditioning type
maxl (int*) flag indicating number of iterations to allow

return value ier is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

notes This routine must be called after the NVECTOR object has been initialized. Allowable values for pretype and maxl are the same as for the C function SUNLinSol_PCG.

The SUNLinSol_PCGSetPrecType and SUNLinSol_PCGSetMaxl routines also support Fortran interfaces for the system and mass matrix solvers.

FSUNPCGSETPRECTYPE

call FSUNPCGSETPRECTYPE(code, pretype, ier)

description The function FSUNPCGSETPRECTYPE can be called for Fortran programs to change the type of preconditioning to use.

arguments code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
pretype (int*) flag indication the type of preconditioning to use.

return value ier is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

notes See SUNLinSol_PCGSetPrecType for complete further documentation of this routine.

FSUNMASSPCGSETPRECTYPE

call FSUNMASSPCGSETPRECTYPE(pretype, ier)

description The function FSUNMASSPCGSETPRECTYPE can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

arguments The arguments are identical to FSUNPCGSETPRECTYPE above, except that code is not needed since mass matrix linear systems only arise in ARKODE.

return value ier is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

notes See SUNLinSol_PCGSetPrecType for complete further documentation of this routine.

FSUNPCGSETMAXL

call FSUNPCGSETMAXL(code, maxl, ier)

description The function FSUNPCGSETMAXL can be called for Fortran programs to change the maximum number of iterations to allow.

arguments code (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
maxl (int*) the number of iterations to allow

return value ier is an int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

notes See SUNLinSol_PCGSetMaxl for complete further documentation of this routine.
8.13 The SUNLinearSolver_PCG implementation

FSUNMASSPCGSETMAXL

Call

FSUNMASSPCGSETMAXL(maxl, ier)

Description
The function FSUNMASSPCGSETMAXL can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

Arguments
The arguments are identical to FSUNPCGSETMAXL above, except that code is not needed since mass matrix linear systems only arise in ARKODE.

Return value
ier is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes
See SUNLinSol_PCGSetMaxl for complete further documentation of this routine.

8.13.2 SUNLINSOL_PCG description

The SUNLINSOL_PCG module defines the content field of a SUNLinearSolver to be the following structure:

```c
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,
The SUNLinearSolver module defines implementations of all “iterative” linear solver operations listed in Sections 8.0.1-8.0.3:

- 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

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.
8.15 SUNLinearSolver functions used by KINSOL

• 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 \( \text{tol} \), 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 \( 10^{*}\text{tol} \), 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 8.3 below, we list the linear solver functions in the SUNLINSOL module used within the KINLS interface in the KINSOL package. In general, KINLS considers two non-overlapping categories of linear solvers: matrix-based and matrix-free, determined based on whether the SUNMATRIX object \( J \) passed to KINSetLinearSolver was not NULL.

Additionally, KINLS will consider a linear solver of either type as iterative if it self-identifies as SUNLINEARSOLVER ITERATIVE (via the SUNLinSolGetType routine). Since both matrix-based and matrix-free linear solvers may be iterative, we only list SUNLINSOL routines that are specifically called based on this type; these routines are in addition to those listed for the other two categories.

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.

The linear solver functions listed below are marked with ✓ to indicate that they are required, or with † to indicate that they are only called if they are non-NULL in the SUNLINSOL implementation that is being used.
Table 8.3: List of linear solver functions usage by KINSOL code modules

<table>
<thead>
<tr>
<th>Function</th>
<th>Matrix-Based</th>
<th>Matrix-Free</th>
<th>Iterative</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNLinSolGetType</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SUNLinSolSetATimes</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUNLinSolSetPreconditioner</td>
<td></td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>SUNLinSolSetScalingVectors</td>
<td>†</td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>SUNLinSolInitialize</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SUNLinSolSetup</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SUNLinSolSolve</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SUNLinSolNumIters</td>
<td></td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>SUNLinSolResNorm</td>
<td></td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>SUNLinSolLastFlag</td>
<td></td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>SUNLinSolFree</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SUNLinSolSpace</td>
<td>†</td>
<td>†</td>
<td></td>
</tr>
</tbody>
</table>

1. **SUNLinSolNumIters** is only used to accumulate overall iterative linear solver statistics. If it is not implemented by the SUNLINSOL module, then KINLS will consider all solves as requiring zero iterations.

2. Although **SUNLinSolResNorm** is optional, if it is not implemented by the SUNLINSOL then KINLS will consider all solves a being exact.

3. Although KINLS does not call **SUNLinSolLastFlag** directly, this routine is available for users to query linear solver issues 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:

  - `solverdir` 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 `solverdir` 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 `solverdir`.

- 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 3.1.3 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. CMake is continually adding new features, and the latest version can be downloaded from [http://www.cmake.org](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 instdir 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:

```bash
% 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 solverdir:

```
% ccmake ../solverdir
```

The default configuration screen is shown in Figure A.1.

![Terminal]

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
```
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 \
  ../solverdir
% 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
A.1 CMake-based installation

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_MINISIZEREL - 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_MINISIZEREL - 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 informa-
tion 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_C_COMPILER - mpicc program
Default:

MPI_CXX_COMPILER - 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 appli-
cations by default without setting any additional configuration options other than MPI_ENABLE.

MPI_Fortran_COMPILER - mpif77 or 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 Fortran77 or Fortran90 examples are enabled
(EXAMPLES_ENABLE_F77 or EXAMPLES_ENABLE_F90 are ON).
MPIEXEC_EXECUTABLE - Specify the executable for running MPI programs
  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_F77_FUNC_CASE - advanced option - Specify the case to use in the Fortran name-mangling scheme, options are: lower or upper
  Default:
  Note: The build system will attempt to infer the Fortran name-mangling scheme using the Fortran compiler. This option should only be used if a Fortran compiler is not available or to override the inferred or default (lower) scheme if one can not be determined. If used, SUNDIALS_F77_FUNC_UNDERSCORES must also be set.

SUNDIALS_F77_FUNC_UNDERSCORES - advanced option - Specify the number of underscores to append in the Fortran name-mangling scheme, options are: none, one, or two
  Default:
  Note: The build system will attempt to infer the Fortran name-mangling scheme using the Fortran compiler. This option should only be used if a Fortran compiler is not available or to override the inferred or default (one) scheme if one can not be determined. If used, SUNDIALS_F77_FUNC_CASE must also be set.

SUNDIALS_INDEX_TYPE - advanced
  Integer type used for SUNDIALS indices. The size must match the size provided for the SUNDIALS_INDEX_SIZE option.
  Default:
  Note: In past SUNDIALS versions, a user could set this option to INT64_T to use 64-bit integers, or INT32_T to use 32-bit integers. Starting in SUNDIALS 3.2.0, these special values are deprecated. For SUNDIALS 3.2.0 and up, a user will only need to use the SUNDIALS_INDEX_SIZE option in most cases.

SUNDIALS_INDEX_SIZE - Integer size (in bits) used for indices in SUNDIALS, options are: 32 or 64
  Default: 64
  Note: The build system tries to find an integer type of appropriate size. Candidate 64-bit integer types are (in order of preference): int64_t, __int64, long long, and long. Candidate 32-bit integers are (in order of preference): int32_t, int, and long. The advanced option, SUNDIALS_INDEX_TYPE can be used to provide a type not listed here.

SUNDIALS_PRECISION - Precision used in SUNDIALS, options are: double, single, or extended
  Default: double
**A.1 CMake-based installation**

**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_SIZE

XSDKPRECISION - 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:

```bash
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DMP'L_ENABLE=ON \
> -DFCMIX_ENABLE=ON \
```
A.1 CMake-based installation

> /home/mynamensundials/solverdir
%
% make install
%

To disable installation of the examples, use:

% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/mynamensundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/mynamensundials/instdir/examples \
> -DMPI_ENABLE=ON \
> -DFCMIX_ENABLE=ON \
> -DEXAMPLES_INSTALL=OFF \
> /home/mynamensundials/solverdir
%
% 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 built 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/mynamensundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/mynamensundials/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/mynamensundials/solverdir
%
% make install
%

When allowing CMake to automatically locate the LAPACK library, CMake may also locate the corresponding BLAS library.

If a working Fortran compiler is not available to infer the Fortran name-mangling scheme, the options SUNDIALS_F77_FUNC_CASE and SUNDIALS_F77_FUNC_UNDERSCORES must be set in order to bypass the check for a Fortran compiler and define the name-mangling scheme. The defaults for these options in earlier versions of sundials were lower and one respectively.
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/solverdir \\
>%
% make install 
```

When allowing CMake to automatically locate the LAPACK library, CMake may also locate the corresponding BLAS library.

If a working Fortran compiler is not available to infer the Fortran name-mangling scheme, the options `SUNDIALS_F77_FUNC_CASE` and `SUNDIALS_F77_FUNC_UNDERSCORES` must be set in order to bypass the check for a Fortran compiler and define the name-mangling scheme. The defaults for these options in earlier versions of `sundials` were `lower` and `one` respectively.

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](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](http://crd-legacy.lbl.gov/~xiaoye/SuperLU/#superlu). `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](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 have been 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 the NVIDIA website: https://developer.nvidia.com/cuda-downloads. To enable CUDA, set CUDA_ENABLE to ON. If CUDA is installed 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

RAJA is a performance portability layer developed by Lawrence Livermore National Laboratory and can be obtained from https://github.com/LLNL/RAJA. SUNDIALS RAJA modules and examples have been tested with RAJA version 0.3. Building SUNDIALS RAJA modules requires a CUDA-enabled RAJA installation. To enable RAJA, set CUDA_ENABLE and RAJA_ENABLE to ON. If RAJA is installed in a nonstandard location you will be prompted to set the variable RAJA_DIR with the path to the RAJA CMake configuration file. To enable building the RAJA examples set EXAMPLES_ENABLE_RAJA to ON.

A.1.5 Testing the build and installation

If SUNDIALS was configured with EXAMPLES_ENABLE_<language> options to ON, then a set of regression tests can be run after building with the make command by running:

```
% make test
```

Additionally, if EXAMPLES_INSTALL was also set to ON, then a set of smoke tests can be run after installing with the make install command by running:

```
% make test_install
```

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 solverdir
2. Create a separate builddir
3. Open a Visual Studio Command Prompt and cd to builddir
4. Run cmake-gui ../solverdir
   (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>
<thead>
<tr>
<th>SHARED</th>
<th>Libraries</th>
<th>n/a</th>
</tr>
</thead>
</table>

continued on next page
### A.4 Installed libraries and exported header files

<table>
<thead>
<tr>
<th>Category</th>
<th>Libraries</th>
<th>Header files</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Header files</strong></td>
<td>sundials/sundials_config.h</td>
<td>nvектор/nvector_serial.h</td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_fconfig.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_types.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_math.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_nvector.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_matrix.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_linear_solver.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_iterative.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_direct.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_dense.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_band.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_nonlinear_solver.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_version.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sundials/sundials_mpi_types.h</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NVVECTOR_SERIAL</th>
<th>Libraries</th>
<th>_header files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>libsundials_nvecserial.lib</td>
<td>nvector/nvector_serial.h</td>
</tr>
<tr>
<td></td>
<td>libsundials_fnvecserial.a</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NVVECTOR_PARALLEL</th>
<th>Libraries</th>
<th>_header files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>libsundials_nvecparallel.lib</td>
<td>nvector/nvector_parallel.h</td>
</tr>
<tr>
<td></td>
<td>libsundials_fnvecparallel.a</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NVVECTOR_OPENMP</th>
<th>Libraries</th>
<th>_header files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>libsundials_nvecopenmp.lib</td>
<td>nvector/nvector_openmp.h</td>
</tr>
<tr>
<td></td>
<td>libsundials_fnvecopenmp.a</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NVVECTOR_PTHREADS</th>
<th>Libraries</th>
<th>_header files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>libsundials_nvecpthreads.lib</td>
<td>nvector/nvector_pthreads.h</td>
</tr>
<tr>
<td></td>
<td>libsundials_fnvecpthreads.a</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NVVECTOR_PARHYP</th>
<th>Libraries</th>
<th>_header files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>libsundials_nvecparhyp.lib</td>
<td>nvector/nvector_parhyp.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NVVECTOR_PETSC</th>
<th>Libraries</th>
<th>_header files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>libsundials_nvecpetsc.lib</td>
<td>nvector/nvector_petsc.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NVVECTOR_CUDA</th>
<th>Libraries</th>
<th>_header files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>libsundials_nveccuda.lib</td>
<td>nvector/nvector_cuda.h</td>
</tr>
<tr>
<td></td>
<td>libsundials_nveccmpicuda.lib</td>
<td>nvector/nvector_mpicuda.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nvector/cuda/ThreadPartitioning.hpp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nvector/cuda/Vector.hpp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nvector/cuda/VectorKernels.cuh</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NVVECTOR_RAJA</th>
<th>Libraries</th>
<th>_header files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>libsundials_nveccudaraaja.lib</td>
<td>nvector/nvector_raja.h</td>
</tr>
<tr>
<td></td>
<td>libsundials_nveccudampiraaja.lib</td>
<td>nvector/nvector_mpiraja.h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nvector/raja/Vector.hpp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUNMATRIX_BAND</th>
<th>Libraries</th>
<th>_header files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>libsundials_ssunmatrixband.lib</td>
<td>nvector/nvector_cuda.h</td>
</tr>
<tr>
<td></td>
<td>libsundials_fssunmatrixband.a</td>
<td>nvector/nvector_mpiraja.h</td>
</tr>
</tbody>
</table>

*continued on next page*
200 SUNDIALS Package Installation Procedure

<table>
<thead>
<tr>
<th>continued from last page</th>
<th>Header files</th>
<th>Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNMATRIX_DENSE</td>
<td>sunmatrix/sunmatrix_band.h</td>
<td>libsundials_sunmatrixdense.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fsunmatrixdense.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_sunmatrixdense.h</td>
</tr>
<tr>
<td>SUNMATRIX_SPARSE</td>
<td>sunmatrix/sunmatrix_sparse.h</td>
<td>libsundials_sunmatrixsparse.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fsunmatrixsparse.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_sunmatrixsparse.h</td>
</tr>
<tr>
<td>SUNLINSOL_BAND</td>
<td>sunlinsol/sunlinsol_band.h</td>
<td>libsundials_sunlinsolband.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fsunlinsolband.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_sunlinsolband.h</td>
</tr>
<tr>
<td>SUNLINSOL_DENSE</td>
<td>sunlinsol/sunlinsoldense.h</td>
<td>libsundials_sunlinsoldense.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fsunlinsoldense.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_sunlinsoldense.h</td>
</tr>
<tr>
<td>SUNLINSOL_KLU</td>
<td>sunlinsol/sunlinsolklu.h</td>
<td>libsundials_sunlinsolklu.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fsunlinsolklu.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsolklu.h</td>
</tr>
<tr>
<td>SUNLINSOL_LAPACKBAND</td>
<td>sunlinsol/sunlinsollapackband.h</td>
<td>libsundials_sunlinsollapackband.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fsunlinsollapackband.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsollapackband.h</td>
</tr>
<tr>
<td>SUNLINSOL_LAPACKDENSE</td>
<td>sunlinsol/sunlinsollapackdense.h</td>
<td>libsundials_sunlinsollapackdense.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fsunlinsollapackdense.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsollapackdense.h</td>
</tr>
<tr>
<td>SUNLINSOL_PCG</td>
<td>sunlinsol/sunlinsolpcg.h</td>
<td>libsundials_sunlinsolpcg.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fsunlinsolpcg.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsolpcg.h</td>
</tr>
<tr>
<td>SUNLINSOL_SPBCGS</td>
<td>sunlinsol/sunlinsolspbcgs.h</td>
<td>libsundials_sunlinsolspbcgs.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fsunlinsolspbcgs.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsolspbcgs.h</td>
</tr>
<tr>
<td>SUNLINSOL_SPFGMR</td>
<td>sunlinsol/sunlinsolspfgmr.h</td>
<td>libsundials_sunlinsolspfgmr.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fsunlinsolspfgmr.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsolspfgmr.h</td>
</tr>
<tr>
<td>SUNLINSOL_SPGMR</td>
<td>sunlinsol/sunlinsolspgmr.h</td>
<td>libsundials_sunlinsolspgmr.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fsunlinsolspgmr.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsolspgmr.h</td>
</tr>
<tr>
<td>SUNLINSOL_SPTFQMR</td>
<td>sunlinsol/sunlinsolsptfqmr.h</td>
<td>libsundials_sunlinsolsptfqmr.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fsunlinsolsptfqmr.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsolsptfqmr.h</td>
</tr>
<tr>
<td>SUNLINSOL_SUPERLUMT</td>
<td>sunlinsol/sunlinsolsuperlumt.h</td>
<td>libsundials_sunlinsolsuperlumt.lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>libsundials_fsunlinsolsuperlumt.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sunlinsol/sunlinsolsuperlumt.h</td>
</tr>
<tr>
<td>SUNNONLINSOL_NEWTON</td>
<td>sunnonlinsol/newton.h</td>
<td>libsundials_sunnonlinsolnewton.lib</td>
</tr>
</tbody>
</table>

continued on next page
### A.4 Installed libraries and exported header files

**SUNNONLINSOL_FIXEDPOINT**
- **Libraries**: 
  - sunnonlinsol/sunnonlinsolfixedpoint.lib
  - sunnonlinsol/sunnonlinsolfixedpoint.a

#### Libraries
- **Libraries**: 
  - sunnonlinsol/sunnonlinsolnewton.a

#### Header files
- **Header files**: 
  - sunnonlinsol/sunnonlinsolnewton.h

---

**CVODE**
- **Libraries**: 
  - cvode/cvode.lib
  - cvode/cvode.a

#### Libraries
- **Libraries**: 
  - cvode/cvode/cvode.h
  - cvode/cvode/cvode_direct.h
  - cvode/cvode/cvode_spils.h
  - cvode/cvode/cvode_bandpre.h

#### Header files
- **Header files**: 
  - cvode/cvode_impl.h
  - cvode/cvode_ls.h
  - cvode/cvode_bandpre.h

---

**CVODES**
- **Libraries**: 
  - cvodes/cvodes.lib

#### Libraries
- **Libraries**: 
  - cvodes/cvodes/cvodes.h
  - cvodes/cvodes/cvodes_direct.h
  - cvodes/cvodes/cvodes_spils.h
  - cvodes/cvodes/cvodes_bandpre.h

#### Header files
- **Header files**: 
  - cvodes/cvodes_impl.h
  - cvodes/cvodes_ls.h
  - cvodes/cvodes_bandpre.h

---

**ARKODE**
- **Libraries**: 
  - arkode/arkode.lib
  - arkode/arkode.a

#### Libraries
- **Libraries**: 
  - arkode/arkode/arkode.h
  - arkode/arkode/arkode_ls.h
  - arkode/arkode/arkode_bandpre.h

#### Header files
- **Header files**: 
  - arkode/arkode_impl.h
  - arkode/arkode_ls.h
  - arkode/arkode_bandpre.h

---

**IDA**
- **Libraries**: 
  - ida/ida.lib
  - ida/ida.a

#### Libraries
- **Libraries**: 
  - ida/ida/ida.h
  - ida/ida/ida_direct.h
  - ida/ida/ida_spils.h
  - ida/ida/ida_bandpre.h

#### Header files
- **Header files**: 
  - ida/ida_impl.h
  - ida/ida_spils.h

---

**IDAS**
- **Libraries**: 
  - idas/idas.lib

#### Libraries
- **Libraries**: 
  - idas/idas/idas.h
  - idas/idas/idas_direct.h
  - idas/idas/idas_spils.h
  - idas/idas/idas_bandpre.h

#### Header files
- **Header files**: 
  - idas/idas_impl.h
  - idas/idas_spils.h
  - idas/idas_bandpre.h

---

**KINSOL**
- **Libraries**: 
  - kinsol/kinsol.lib
  - kinsol/kinsol.a

#### Libraries
- **Libraries**: 
  - kinsol/kinsol/kinsol.h
  - kinsol/kinsol/kinsol_direct.h
  - kinsol/kinsol/kinsol_spils.h
  - kinsol/kinsol/kinsol_bandpre.h

#### Header files
- **Header files**: 
  - kinsol/kinsol_impl.h
  - kinsol/kinsol_ls.h
  - kinsol/kinsol_spils.h
  - kinsol/kinsol_bandpre.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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KIN_ETACHoice1</td>
<td>1</td>
<td>Use Eisenstat and Walker Choice 1 for $\eta$.</td>
</tr>
<tr>
<td>KIN_ETACHoice2</td>
<td>2</td>
<td>Use Eisenstat and Walker Choice 2 for $\eta$.</td>
</tr>
<tr>
<td>KIN_ETACONSTANT</td>
<td>3</td>
<td>Use constant value for $\eta$.</td>
</tr>
<tr>
<td>KIN_NONE</td>
<td>0</td>
<td>Use inexact Newton globalization.</td>
</tr>
<tr>
<td>KIN_LINESEARCH</td>
<td>1</td>
<td>Use linesearch globalization.</td>
</tr>
</tbody>
</table>

Iterative linear solver modules

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREC_NONE</td>
<td>0</td>
<td>No preconditioning</td>
</tr>
<tr>
<td>PREC_RIGHT</td>
<td>2</td>
<td>Preconditioning on the right.</td>
</tr>
<tr>
<td>MODIFIED_GS</td>
<td>1</td>
<td>Use modified Gram-Schmidt procedure.</td>
</tr>
<tr>
<td>CLASSICAL_GS</td>
<td>2</td>
<td>Use classical Gram-Schmidt procedure.</td>
</tr>
</tbody>
</table>

B.2 KINSOL output constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KIN_SUCCESS</td>
<td>0</td>
<td>Successful function return.</td>
</tr>
<tr>
<td>KIN_INITIAL_GUESS_OK</td>
<td>1</td>
<td>The initial user-supplied guess already satisfies the stopping criterion.</td>
</tr>
<tr>
<td>KIN_STEP_LT_STPTOL</td>
<td>2</td>
<td>The stopping tolerance on scaled step length was satisfied.</td>
</tr>
<tr>
<td>KIN_WARNING</td>
<td>99</td>
<td>A non-fatal warning. The solver will continue.</td>
</tr>
<tr>
<td>KIN_MEM_NULL</td>
<td>-1</td>
<td>The kin_mem argument was NULL.</td>
</tr>
<tr>
<td>KIN_ILL_INPUT</td>
<td>-2</td>
<td>One of the function inputs is illegal.</td>
</tr>
<tr>
<td>KIN_NO_MALLOC</td>
<td>-3</td>
<td>The KINSOL memory was not allocated by a call to KINMalloc.</td>
</tr>
<tr>
<td>Constant</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>KIN_MEM_FAIL</td>
<td>-4</td>
<td>A memory allocation failed.</td>
</tr>
<tr>
<td>KIN_LINESEARCH_NONCONV</td>
<td>-5</td>
<td>The linesearch algorithm was unable to find an iterate sufficiently distinct from the current iterate.</td>
</tr>
<tr>
<td>KIN_MAXITER_REACHED</td>
<td>-6</td>
<td>The maximum number of nonlinear iterations has been reached.</td>
</tr>
<tr>
<td>KIN_MXNEWT_5X_EXCEEDED</td>
<td>-7</td>
<td>Five consecutive steps have been taken that satisfy a scaled step length test.</td>
</tr>
<tr>
<td>KIN_LINESEARCH_BCFAIL</td>
<td>-8</td>
<td>The linesearch algorithm was unable to satisfy the $\beta$-condition for $\text{nbcfails}$ iterations.</td>
</tr>
<tr>
<td>KIN_LINSOLV_NO_RECOVERY</td>
<td>-9</td>
<td>The user-supplied routine preconditioner $\text{slve}$ function failed recoverably, but the preconditioner is already current.</td>
</tr>
<tr>
<td>KIN_LINIT_FAIL</td>
<td>-10</td>
<td>The linear solver’s initialization function failed.</td>
</tr>
<tr>
<td>KIN_LSETUP_FAIL</td>
<td>-11</td>
<td>The linear solver’s setup function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>KIN_LSOLVE_FAIL</td>
<td>-12</td>
<td>The linear solver’s solve function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>KIN_SYSFUNC_FAIL</td>
<td>-13</td>
<td>The system function failed in an unrecoverable manner.</td>
</tr>
<tr>
<td>KIN_FIRST_SYSFUNC_ERR</td>
<td>-14</td>
<td>The system function failed recoverably at the first call.</td>
</tr>
<tr>
<td>KIN_REPTD_SYSFUNC_ERR</td>
<td>-15</td>
<td>The system function had repeated recoverable errors.</td>
</tr>
<tr>
<td>KINLS_SUCCESS</td>
<td>0</td>
<td>Successful function return.</td>
</tr>
<tr>
<td>KINLS_MEM_NULL</td>
<td>-1</td>
<td>The $\text{kin_mem}$ argument was $\text{NULL}$.</td>
</tr>
<tr>
<td>KINLS_LMEM_NULL</td>
<td>-2</td>
<td>The KINLS linear solver has not been initialized.</td>
</tr>
<tr>
<td>KINLS_ILL_INPUT</td>
<td>-3</td>
<td>The KINLS solver is not compatible with the current NVECTOR module, or an input value was illegal.</td>
</tr>
<tr>
<td>KINLS_MEM_FAIL</td>
<td>-4</td>
<td>A memory allocation request failed.</td>
</tr>
<tr>
<td>KINLS_PMEM_NULL</td>
<td>-5</td>
<td>The preconditioner module has not been initialized.</td>
</tr>
<tr>
<td>KINLS_JACFUNC_ERR</td>
<td>-6</td>
<td>The Jacobian function failed</td>
</tr>
<tr>
<td>KINLS_SUNMAT_FAIL</td>
<td>-7</td>
<td>An error occurred with the current SUNMATRIX module.</td>
</tr>
<tr>
<td>KINLS_SUNLINS_FAIL</td>
<td>-8</td>
<td>An error occurred with the current SUNLINSOL module.</td>
</tr>
</tbody>
</table>

**KINLS linear solver interface**
Bibliography


Anderson acceleration
  definition, 18
Anderson acceleration UA
  definition, 18
BIG_REAL, 26, 81
booleantype, 26
data types
  Fortran, 63
eh_data, 53
ero message
  user-defined handler, 36
error messages, 34
  redirecting, 36
  user-defined handler, 53
ETA_CONST, 71
ETA_FORM, 71
ETA_PARAMS, 71
Fixed-point iteration
  definition, 17
fixed-point system
  definition, 13
FKBJAC, 68
FKCOMM, 74
FKDJAC, 68
FKFUN, 65
FKINBANDSETJAC, 68
FKINBBBD interface module
  interface to the KINBBBD module, 71
FKINBBBDINIT, 73
FKINBBDOPT, 74
FKINCREATE, 67
FKINDESESETCAC, 68
FKINDESESETJAC, 68
FKINDESESETJAC, 68
FKINFREE, 70
FKININIT, 67
FKINJTIMES, 69, 74
FKINJSETUP, 74
FKINLSINIT, 67
FKINLSSETJAC, 69, 73
FKINLSSETPREC, 70
FKINSETIIN, 71
FKINSETIIN, 71
FKINSETIIN, 71
FKINSETVIN, 71
FKINSOL, 70
FKINSOL interface module
  interface to the KINBBBD module, 74
  optional input and output, 71
  usage, 65–70
  user-callable functions, 64–65
  user-supplied functions, 65
FKINSPARSESETJAC, 69
FKINSPILSETJAC, 69
FKINSPILSETPREC, 70
FKINSPILSINIT, 67
FKINSPJAC, 68
FKLOCFN, 74
FKPSET, 70
FKPSOL, 69
FNORM_TOL, 71
FSUNBANDLINSOLINIT, 140
FSUNDENSELINSOLINIT, 138
FSUNKLUINIT, 148
FSUNKLUREINIT, 148
FSUNKLUSETORDERING, 149
FSUNLAPACKBANDINIT, 145
FSUNLAPACKDENSEINIT, 142
FSUNMASSBANDLINSOLINIT, 141
FSUNMASSDENSELINSOLINIT, 139
FSUNMASSKLUINIT, 148
FSUNMASSKLUREINIT, 149
FSUNMASSKLUSETORDERING, 149
FSUNMASSSLAPACKBANDINIT, 145
FSUNMASSSLAPACKDENSEINIT, 143
FSUNMASSPCGINIT, 180
FSUNMASSPCGSETMAXL, 181
FSUNMASSPCGSETPRECYPE, 180
FSUNMASSPCGSETPRECYPE, 180
FSUNMASSPCGSETPRECYPE, 180
FSUNMASSPFQMRINIT, 164
FSUNMASSPFQMRSETGSTYPE, 164
FSUNMASSPFQMRSETMAXRS, 165
FSUNMASSPFQMRSETPRECTYPE, 165
Jacobian times vector
difference quotient, 44
user-supplied, 44, 56
sparse
use in FKINSOL, 68
user-supplied, 44, 54–55

KIN_ETACHOICE1, 39
KIN_ETACHOICE2, 39
KIN_ETACONSTANT, 39
KIN_FIRST_SYSFUNC_ERR, 34
KIN_FP, 33
KIN_ILL_INPUT, 31, 33, 37–43
KIN_INITIAL_GUESS_OK, 33
KIN_LINESEARCH, 33
KIN_LINESEARCH_BCFAIL, 34
KIN_LINESEARCH_NONCONV, 33
KIN_LINIT_FAIL, 34
KIN_LINSOLV_NO_RECOVERY, 34
KIN_LSETUP_FAIL, 34
KIN_LSOLVE_FAIL, 34
KIN_MAXITER_REACHED, 34
KIN_MEM_FAIL, 31, 33
KIN_MEM_NULL, 31, 33, 36–43, 47–49
KIN_MXNEWT_5X_EXCEEDED, 34
KIN_NO_MALLOC, 33
KIN_NONE, 33
KIN_PICARD, 33
KIN_REPTD_SYSFUNC_ERR, 34
KIN_STEP_LT_STPTOL, 33
KIN_SUCCESS, 31, 33, 36–43, 47–49
KIN_SYSFUNC_FAIL, 34
KIN_WARNING, 53

KINBBDPRE preconditioner
optional output, 61–62
usage, 59–60
user-callable functions, 60–61
user-supplied functions, 58–59
KINBBDPrecGetNumGfnEvals, 61
KINBBDPrecGetWorkSpace, 61
KINBBDPrecInit, 60
KINCreate, 31
KINDlsGetLastFlag, 52
KINDlsGetNumFuncEvals, 50
KINDlsGetNumJacEvals, 50
KINDlsGetReturnFlagName, 52
KINDlsGetWorkspace, 49
KINDlsJacFn, 55
KINDlsSetJacFn, 44
KINDlsSetLinearSolver, 32
KINErrHandlerFn, 53
KINFree, 31
KINGetFuncNorm, 48
KINGetLastLinFlag, 52

half-bandwidths, 60
header files, 27, 59
ih_data, 54
Inexact Newton iteration
definition, 13
info message
user-defined handler, 36
info messages
redirecting, 36
informational messages
user-defined handler, 54
IOUT, 71, 72

Jacobian approximation function
band
use in FKINSOL, 68
dense
use in FKINSOL, 68
difference quotient, 44
INDEX

KINGetLinReturnFlagName, 52
KINGetLinWorkSpace, 49
KINGetNumBacktrackOps, 48
KINGetNumBetaCondFails, 48
KINGetNumFuncEvals, 47
KINGetNumJacEvals, 49
KINGetNumJtimesEvals, 51
KINGetNumLinConvFails, 50
KINGetNumLinFuncEvals, 50
KINGetNumLinIters, 50
KINGetNumLinSolvIters, 48
KINGetNumLinTimesEvals, 51
KINGetNumPrecEvals, 51
KINGetNumPrecSolves, 51
KINGetStepLength, 49
KINGetWorkSpace, 47
KINInfoHandlerFn, 54
KINInit, 31, 44
KINLMEM_NULL, 50
KINLS linear solver
  Jacobian-vector product approximation used by, 44
  memory requirements, 49
KINLS linear solver interface
  Jacobian approximation used by, 44
  optional input, 44–46
  optional output, 49–52
  preconditioner setup function, 45, 57
  preconditioner solve function, 45, 56
  use in KINSOL, 67
KINLS_ILL_INPUT, 32, 61
KINLS_LMEM_NULL, 44, 45, 49–52, 61
KINLS_MEM_FAIL, 32, 61
KINLS_MEM_NULL, 32, 44, 45, 49–52
KINLS_PMEM_NULL, 61
KINLS_SUCCESS, 32, 44, 45, 49–52
KINLS_SUNLS_FAIL, 32, 45
KINLsJacFn, 54
KINLsJacTimesVecFn, 56
KINLsPrecSetupFn, 57
KINLsPrecSolveFn, 56
KINSetConstraints, 42
KINSetErrFile, 36
KINSetErrHandlerFn, 36
KINSetEtaConstValue, 40
KINSetEtaForm, 39
KINSetEtaParams, 40
KINSetFuncNormTol, 42
KINSetInfoFile, 36
KINSetInfoHandlerFn, 37
KINSetJacFn, 44
KINSetJacTimesVecFn, 45
KINSetLinearSolver, 29, 32, 54, 126, 183
KINSetMAA, 43
KINSetMaxBetaFails, 41
KINSetMaxNewtonStep, 41
KINSetMaxSetupCalls, 39
KINSetMaxNumSetupCalls, 39
KINSetNoInitSetup, 38
KINSetNoMinEps, 41
KINSetNoResMon, 38
KINSetNumMaxIters, 38
KINSetPreconditioner, 45
KINSetPrintLevel, 37
KINSetRelErrFunc, 42
KINSetResMonConstValue, 40
KINSetResMonParams, 40
KINSetScaledStepTol, 42
KINSetSysFunc, 43
KINSetUserData, 37

KINSOL
  brief description of, 1
  motivation for writing in C, 2
  package structure, 21
  relationship to NKSOL, 1
KINSOL linear solver interface
KINLS, 32
KINSOL linear solver interfaces, 21
KINSOL linear solvers
  header files, 27
  implementation details, 22
  NVECTOR compatibility, 25
  selecting one, 32
KINsol, 29, 33
knsol/knsol.h, 27
knsol/knsol_ls.h, 27
KINSOL/KINSOL linear solvers
  selecting one, 31
KINSpilsGetLastFlag, 52
KINSpilsGetNumConvFails, 51
KINSpilsGetNumFuncEvals, 50
KINSpilsGetNumJtimesEval, 52
KINSpilsGetNumLinIters, 50
KINSpilsGetNumLinSolvIters, 48
KINSpilsGetNumPrecEvals, 51
KINSpilsGetNumPrecSolves, 51
KINSpilsGetReturnFlagName, 52
KINSpilsGetWorkspace, 49
KINSpilsJacTimesVecFn, 56
KINSpilsPrecSetupFn, 58
KINSpilsPrecSolveFn, 57
KINSpilsSetJacTimesVecFn, 45
KINSpilsSetLinearSolver, 32
KINSpilsSetPreconditioner, 46
KINSysFn, 31, 53

MAA, 71
MAX_NITERS, 71
MAX_SETUPS, 71
MAX_SP_SETUPS, 71
MAX_STEP, 71
memory requirements
  KINBBDPRE preconditioner, 61
  KINLS linear solver, 49
  KINSL solver, 47
Modified Newton iteration
  definition, 13
N_VCloneVectorArray, 76
N_VCloneVectorArray_OpenMP, 91
N_VCloneVectorArray_Parallel, 89
N_VCloneVectorArray_ParHyp, 95
N_VCloneVectorArray_Pthreads, 97
N_VCloneVectorArray_Serial, 86
N_VCloneVectorArrayEmpty, 76
N_VCloneVectorArrayEmpty_OpenMP, 91
N_VCloneVectorArrayEmpty_Parallel, 89
N_VCloneVectorArrayEmpty_ParHyp, 96
N_VCloneVectorArrayEmpty_Petsc, 97
N_VCloneVectorArrayEmpty_Pthreads, 94
N_VCloneVectorArrayEmpty_Serial, 86
N_VCopyToDevice_Cuda, 100
N_VCopyToDevice_Raja, 103
N_VCopyToDevice_Serial, 103
N_VDestroyVectorArray, 76
N_VDestroyVectorArray_OpenMP, 91
N_VDestroyVectorArray_Parallel, 89
N_VDestroyVectorArray_ParHyp, 96
N_VDestroyVectorArray_Petsc, 97
N_VDestroyVectorArray_Pthreads, 94
N_VDestroyVectorArray_Serial, 86
N_Vector, 27, 75
N_VGetDeviceArrayPointer_Cuda, 100
N_VGetDeviceArrayPointer_Raja, 103
N_VGetHostArrayPointer_Cuda, 100
N_VGetHostArrayPointer_Raja, 103
N_VGetLength_Cuda, 100
N_VGetLength_OpenMP, 92
N_VGetLength_Parallel, 89
N_VGetLength_Pthreads, 94
N_VGetLength_Raja, 103
N_VGetLength_Serial, 86
N_VGetLocalLength_Parallel, 89
N_VGetVector_ParHyp, 95
N_VMake_Petsc, 97
N_VMake_Raja, 102
N_VMake_Serial, 86
N_VNew_CUDA, 99
N_VNew_OpenMP, 91
N_VNew_Parallel, 88
N_VNew_Pthreads, 93
N_VNew_Raja, 102
N_VNew_Serial, 86
N_VNewEmpty_Cuda, 100
N_VNewEmpty_OpenMP, 91
N_VNewEmpty_Parallel, 88
N_VNewEmpty_ParHyp, 95
N_VNewEmpty_Petsc, 97
N_VNewEmpty_Pthreads, 93
N_VNewEmpty_Raja, 102
N_VNewEmpty_Serial, 86
N_VPrint_Cuda, 100
N_VPrint_OpenMP, 92
N_VPrint_Parallel, 89
N_VPrint_ParHyp, 96
N_VPrint_Petsc, 97
N_VPrint_Pthreads, 94
N_VPrint_Raja, 103
N_VPrint_Serial, 86
N_VPrintFile_Cuda, 100
N_VPrintFile_OpenMP, 92
N_VPrintFile_Parallel, 89
N_VPrintFile_ParHyp, 96
N_VPrintFile_Petsc, 97
N_VPrintFile_Pthreads, 94
N_VPrintFile_Raja, 103
N_VPrintFile_Serial, 86
NO_INIT_SETUP, 71
NO_MIN_EPS, 71
NO_RES_MON, 71
nonlinear system
  definition, 13
NV_COMM_P, 88
NV_CONTENTOMP, 90
NV_CONTENT_P, 87
NV_CONTENT_PT, 93
NV_CONTENTS, 85
NV_DATAOMP, 90
NV_DATAP, 87
NV_DATAPT, 93
NV_DATAS, 85
NV_GLOBLNGTH, 90
NV_IthOMP, 91
NV_IthP, 88
NV_IthPT, 93
NV_IthS, 85
NV_LENGTHOMP, 90
NV_LENGTHPT, 93
NV_LENGTHS, 85
NV_LOCLENGTHP, 87
optional input
  generic linear solver interface, 44–46
  solver, 36–44

optional output
  band-block-diagonal preconditioner, 61–62
  generic linear solver interface, 49–52
  solver, 47–49
  version, 46–47

Picard iteration
  definition, 17
  portability, 26
  Fortran, 63

Preconditioner setup routine
  use in FKINSOL, 69

Preconditioner solve routine
  use in FKINSOL, 69

preconditioning
  advice on, 22
  setup and solve phases, 22
  user-supplied, 45–46, 56, 57

PRINT_LEVEL, 71
  problem-defining function, 53

RCONST, 26
  realtype, 26
  RERR_FUNC, 71
  RMON_CONST, 71
  RMON_PARAMS, 71
  ROUT, 71, 72

SM_COLS_B, 117
  SM_COLS_D, 113
  SM_COLUMN_B, 55, 117
  SM_COLUMN_D, 55, 113
  SM_COLUMN_ELEMENT_B, 55, 117
  SM_COLUMNS_B, 117
  SM_COLUMNS_D, 112
  SM_COLUMNS_S, 123
  SM_CONTENT_B, 117
  SM_CONTENT_D, 112
  SM_CONTENT_S, 121
  SM_DATA_B, 117
  SM_DATA_D, 113
  SM_DATA_S, 123
  SM_ELEMENT_B, 55, 117
  SM_ELEMENT_D, 55, 113

SM_INDEXPTRS_S, 123
  SM_INDEXVALS_S, 123
  SM_LBAND_B, 117
  SM_LDATA_B, 117
  SM_LDATA_D, 112
  SM_LDIM_B, 117
  SM_NNZ_S, 55, 123
  SM_NP_S, 123
  SM_ROWS_B, 117
  SM_ROWS_D, 112
  SM_ROWS_S, 123
  SM_SPARSETYPE_S, 123
  SM_SUBAND_B, 117
  SM UBAND_B, 117
  SMALL_REAL, 26
  SSTEP_TOL, 71

SUNBandLinearSolver, 140
  SUNBandMatrix, 29, 118
  SUNBandMatrix_Cols, 119
  SUNBandMatrix_Column, 119
  SUNBandMatrix_Columns, 118
  SUNBandMatrix_Data, 119
  SUNBandMatrix_LDim, 119
  SUNBandMatrix_LowerBandwidth, 118
  SUNBandMatrix_Print, 118
  SUNBandMatrix_Rows, 118
  SUNBandMatrix_StoredUpperBandwidth, 118
  SUNBandMatrix_UpperBandwidth, 118
  SUNDenseLinearSolver, 138
  SUNDenseMatrix, 29, 113
  SUNDenseMatrix_Cols, 114
  SUNDenseMatrix_Column, 114
  SUNDenseMatrix_Columns, 114
  SUNDenseMatrix_Data, 114
  SUNDenseMatrix_LData, 114
  SUNDenseMatrix_Print, 113
  SUNDenseMatrix_Rows, 114

sundials/sundials_linear_solver.h, 129
  sundials_nvector.h, 27
  sundials_types.h, 26, 27
  SUNDIALSGetVersion, 46
  SUNDIALSGetVersionNumber, 47
  sunindextype, 26
  SUNKLU, 148
  SUNKLUReInit, 148
  SUNKLUSetOrdering, 148
  SUNLapackBand, 144
  SUNLapackDense, 142
  SUNLinearSolver, 129
  SUNLinearSolver, 136
  SUNLinearSolver module, 129
  sunlinsol/sunlinsol_band.h, 27
  sunlinsol/sunlinsol_dense.h, 27
  sunlinsol/sunlinsol_klu.h, 27
sunlinsol/sunlinsol_lapackband.h, 27
sunlinsol/sunlinsol_lapackdense.h, 27
sunlinsol/sunlinsol_pcg.h, 27
sunlinsol/sunlinsol_spbcs.h, 27
sunlinsol/sunlinsol_spfgmr.h, 27
sunlinsol/sunlinsol_spgr.h, 27
sunlinsol/sunlinsol_sptfqmr.h, 27
sunlinsol/sunlinsol_superlumt.h, 27
SUNLinSol_Band, 32, 140
SUNLinSol_Dense, 32, 138
SUNLinSol_KLU, 32, 146
SUNLinSol_KLUReInit, 147
SUNLinSol_LapackBand, 32, 144
SUNLinSol_LapackDense, 32, 142
SUNLinSol_PCG, 32, 178–180
SUNLinSol_PCGSetMaxl, 179
SUNLinSol_PCGSetPrecType, 179
SUNLinSol_SPBGS, 32, 168, 169
SUNLinSol_SPBCGSMaxl, 168, 169
SUNLinSol_SPBCGSSetMaxl, 168, 169
SUNLinSol_SPBCGSSetPrecType, 168, 169
SUNLinSol_SPFGMR, 32, 162–164
SUNLinSol_SPFGMRSetGSType, 163
SUNLinSol_SPFGMRSetMaxRestarts, 163
SUNLinSol_SPFGMRSetPrecType, 162, 163
SUNLinSol_SPGMR, 32, 156–158
SUNLinSol_SPGMRSetGSType, 157
SUNLinSol_SPGMRSetMaxRestarts, 157
SUNLinSol_SPGMRSetPrecType, 156, 157
SUNLinSol_SPTFQMR, 32, 173, 174
SUNLinSol_SPTFQMRSetMaxl, 173, 174
SUNLinSol_SPTFQMRSetPrecType, 173, 174
SUNLinSol_SuperLUMT, 32, 152
SUNLinSol_SuperLUMTSetOrdering, 152, 153
SUNLinSol_Free, 30, 130, 131
SUNLinSolGetType, 130, 183
SUNLinSol_Initialize, 130
SUNLinSol_LastFlag, 133
SUNLinSol_NumIters, 133
SUNLinSol_ResNorm, 133
SUNLinSol_SetATimes, 132
SUNLinSol_SetPreconditioner, 132
SUNLinSol_SetScalingVectors, 132
SUNLinSol_Setup, 130, 131
SUNLinSol_Solve, 130, 131
SUNLinSol_Space, 133
SUNMatDestroy, 30
SUNMatrix, 109
SUNMatrix module, 109
SUNNONLINEARSOLVER_DIRECT, 130
SUNNONLINEARSOLVER_ITERATIVE, 130
SUNPCG, 179
SUNPCGSetMaxl, 179
SUNPCGSetPrecType, 179
SUNSparseFromBandMatrix, 124
SUNSparseFromDenseMatrix, 123
SUNSparseMatrix, 29, 123
SUNSparseMatrix_Columns, 124
SUNSparseMatrix_Data, 125
SUNSparseMatrix_IndexPointers, 125
SUNSparseMatrix_IndexValues, 125
SUNSparseMatrix_NNZ, 55, 125
SUNSparseMatrix_NP, 125
SUNSparseMatrix_Print, 124
SUNSparseMatrix_Realloc, 124
SUNSparseMatrix_Rows, 124
SUNSparseMatrix_SparseType, 125
SUNSBCGS, 169
SUNSBCGSSetMaxl, 169
SUNSBCGSSetPrecType, 169
SUNSPFGMR, 163
SUNSPFGMRS GSType, 163
SUNSPFGMRSMaxRestarts, 163
SUNSPFGMRSetPrecType, 163
SUNSPGMR, 157
SUNSPGMRSetGSType, 157
SUNSPGMRSetMaxRestarts, 157
SUNSPGMRSetPrecType, 157
SUNSPTFQMR, 174
SUNSPTFQMRSetMaxl, 174
SUNSPTFQMRSetPrecType, 174
SUNSPLUMT, 152
SUNSPLUMTSetOrdering, 152
UNIT_ROUND OFF, 26
User main program
FKINBBDD usage, 73
FKINSOL usage, 65
KINBBDPRE usage, 59
KINSOL usage, 28
user_data, 53, 59