

User Documentation for KINSOL v5.8.0
(SUNDIALS v5.8.0)

Alan C. Hindmarsh¹, Radu Serban¹, Cody J. Balos¹,
David J. Gardner¹, Daniel R. Reynolds², and Carol S. Woodward¹

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

²*Department of Mathematics, Southern Methodist University*

September 30, 2021



UCRL-SM-208116

DISCLAIMER

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

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

CONTRIBUTORS

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

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

Contents

List of Tables	ix
List of Figures	xi
1 Introduction	1
1.1 Historical Background	1
1.2 Changes from previous versions	2
1.3 Reading this User Guide	15
1.4 SUNDIALS Release License	16
2 Mathematical Considerations	19
3 Code Organization	27
3.1 SUNDIALS organization	27
3.2 KINSOL organization	28
4 Using KINSOL for C Applications	31
4.1 Access to library and header files	31
4.2 Data types	32
4.3 Header files	33
4.4 A skeleton of the user's main program	34
4.5 User-callable functions	36
4.6 User-supplied functions	61
4.7 A parallel band-block-diagonal preconditioner module	66
5 Using KINSOL for Fortran Applications	71
5.1 KINSOL Fortran 2003 Interface Module	71
5.2 Important note on portability	77
5.3 Fortran Data Types	77
6 KINSOL Features for GPU Accelerated Computing	89
6.1 SUNDIALS GPU Programming Model	89
6.2 Steps for Using GPU Accelerated SUNDIALS	90
7 Description of the NVECTOR module	91
7.1 The NVECTOR API	91
7.2 NVECTOR functions used by KINSOL	112
7.3 The NVECTOR_SERIAL implementation	114
7.4 The NVECTOR_PARALLEL implementation	119
7.5 The NVECTOR_OPENMP implementation	124
7.6 The NVECTOR_PTHREADS implementation	130
7.7 The NVECTOR_PARHYP implementation	135
7.8 The NVECTOR_PETSC implementation	139

7.9	The NVECTOR_CUDA implementation	142
7.10	The NVECTOR_HIP implementation	149
7.11	The NVECTOR_RAJA implementation	155
7.12	The NVECTOR_SYCL implementation	159
7.13	The NVECTOR_OPENMPDEV implementation	165
7.14	The NVECTOR_TRILINOS implementation	170
7.15	The NVECTOR_MANYVECTOR implementation	171
7.16	The NVECTOR_MPIMANYVECTOR implementation	176
7.17	The NVECTOR_MPIPLUSX implementation	181
7.18	NVECTOR Examples	182
8	Description of the SUNMatrix module	187
8.1	The SUNMatrix API	187
8.2	SUNMatrix functions used by KINSOL	193
8.3	The SUNMatrix_Dense implementation	194
8.4	The SUNMatrix_Band implementation	197
8.5	The SUNMatrix_Sparse implementation	204
8.6	The SUNMatrix_SLUNRloc implementation	211
8.7	The SUNMatrix_cuSparse implementation	213
8.8	The SUNMATRIX_MAGMADENSE implementation	218
8.9	The SUNMATRIX_ONEMKLDENSE implementation	221
9	Description of the SUNLinearSolver module	227
9.1	The SUNLinearSolver API	228
9.2	Compatibility of SUNLinearSolver modules	237
9.3	Implementing a custom SUNLinearSolver module	238
9.4	KINSOL SUNLinearSolver interface	240
9.5	The SUNLinearSolver_Dense implementation	242
9.6	The SUNLinearSolver_Band implementation	245
9.7	The SUNLinearSolver_LapackDense implementation	247
9.8	The SUNLinearSolver_LapackBand implementation	250
9.9	The SUNLinearSolver_KLU implementation	252
9.10	The SUNLinearSolver_SuperLUDIST implementation	259
9.11	The SUNLinearSolver_SuperLUMT implementation	263
9.12	The SUNLinearSolver_cuSolverSp_batchQR implementation	267
9.13	The SUNLinearSolver_MagmaDense implementation	269
9.14	The SUNLinearSolver_OneMklDense Implementation	271
9.15	The SUNLinearSolver_SPGMR implementation	272
9.16	The SUNLinearSolver_SPGFMR implementation	279
9.17	The SUNLinearSolver_SPBCGS implementation	287
9.18	The SUNLinearSolver_SPTFQMR implementation	293
9.19	The SUNLinearSolver_PCG implementation	299
9.20	SUNLinearSolver Examples	306
10	Description of the SUNMemory module	309
10.1	The SUNMemoryHelper API	309
10.2	The SUNMemoryHelper_Cuda implementation	313
10.3	The SUNMemoryHelper_Hip implementation	314
10.4	The SUNMemoryHelper_Sycl implementation	315
A	SUNDIALS Package Installation Procedure	319
A.1	CMake-based installation	320
A.2	Building and Running Examples	332
A.3	Configuring, building, and installing on Windows	332
A.4	Installed libraries and exported header files	333

B KINSOL Constants	341
B.1 KINSOL input constants	341
B.2 KINSOL output constants	341
C SUNDIALS Release History	343
Bibliography	345
Index	349

List of Tables

4.1	SUNDIALS linear solver interfaces and vector implementations that can be used for each.	36
4.2	Optional inputs for KINSOL and KINLS	41
4.3	Optional outputs from KINSOL and KINLS	55
5.1	Summary of Fortran 2003 interfaces for shared SUNDIALS modules.	72
5.2	C/Fortran 2003 Equivalent Types	73
5.3	Keys for setting FKINSOL optional inputs	85
5.4	Description of the FKINSOL optional output arrays IOUT and ROUT	86
6.1	List of SUNDIALS GPU Enabled Modules.	90
7.1	Vector Identifications associated with vector kernels supplied with SUNDIALS.	108
7.2	List of vector functions usage by KINSOL code modules	113
8.1	Description of the SUNMatrix return codes	190
8.2	Identifiers associated with matrix kernels supplied with SUNDIALS.	191
8.3	SUNDIALS matrix interfaces and vector implementations that can be used for each. . .	191
8.4	List of matrix functions usage by KINSOL code modules	193
9.1	Description of the SUNLinearSolver error codes	235
9.2	SUNDIALS matrix-based linear solvers and matrix implementations that can be used for each.	237
9.3	List of linear solver function usage in the KINLS interface	241
A.1	SUNDIALS libraries and header files	335
C.1	Release History	343

List of Figures

3.1	High-level diagram of the SUNDIALS suite.	27
3.2	Directory structure of the SUNDIALS source tree.	28
3.3	Overall structure diagram of the KINSOL package	29
8.1	Diagram of the storage for a SUNMATRIX_BAND object	199
8.2	Diagram of the storage for a compressed-sparse-column matrix	206
A.1	Initial <i>ccmake</i> configuration screen	321
A.2	Changing the <i>instdir</i>	322

Chapter 1

Introduction

KINSOL is part of a software family called SUNDIALS: SUite of Nonlinear and Differential/ALgebraic equation Solvers [26]. 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 [13]. 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) [38], FGMRES (Flexible Generalized Minimum RESidual) [37], Bi-CGStab (Bi-Conjugate Gradient Stabilized) [40], TFQMR (Transpose-Free Quasi-Minimal Residual) [23], and PCG (Preconditioned Conjugate Gradient) [25] linear iterative methods. As Krylov methods, these require little matrix storage for solving the Newton equations as compared to direct methods. However, the algorithms allow for a user-supplied preconditioner matrix, and, for most problems, preconditioning is essential for an efficient solution. For very large nonlinear algebraic systems, the Krylov methods are preferable over direct linear solver methods, and are often the only feasible choice. Among the Krylov methods in SUNDIALS, we recommend GMRES as the best overall choice. However, users are encouraged to compare all three, especially if encountering convergence failures with GMRES. Bi-CGStab and TFQMR have an advantage in storage requirements, in that the number of workspace vectors they require is fixed, while that number for GMRES depends on the desired Krylov subspace size. FGMRES has an advantage in that it is designed to support preconditioners that vary between iterations (e.g. iterative methods). PCG exhibits rapid convergence and minimal workspace vectors, but only works for symmetric linear systems.

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

user-supplied or generated internally by difference quotients. KINSOL also includes interfaces to the sparse direct solvers KLU [16, 3], and the threaded sparse direct solver, SuperLU_MT [31, 18, 9].

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 v5.8.0

The RAJA NVECTOR implementation has been updated to support the SYCL backend in addition to the CUDA and HIP backend. Users can choose the backend when configuring SUNDIALS by using the `SUNDIALS_RAJA_BACKENDS` CMake variable. This module remains experimental and is subject to change from version to version.

A new SUNMATRIX and SUNLINSOL implementation were added to interface with the Intel oneAPI Math Kernel Library (oneMKL). Both the matrix and the linear solver support general dense linear systems as well as block diagonal linear systems. See Chapter 9.14 for more details. This module is experimental and is subject to change from version to version.

Added a new *optional* function to the SUNLinearSolver API, `SUNLinSolSetZeroGuess`, to indicate that the next call to `SUNLinSolSolve` will be made with a zero initial guess. SUNLinearSolver implementations that do not use the `SUNLinSolNewEmpty` constructor will, at a minimum, need set the `setzeroguess` function pointer in the linear solver `ops` structure to `NULL`. The SUNDIALS iterative linear solver implementations have been updated to leverage this new set function to remove one dot product per solve.

New KINSOL options have been added to apply a constant damping in the fixed point and Picard iterations (see `KINSetDamping`), to delay the start of Anderson acceleration with the fixed point and Picard iterations (see `KINSetDelayAA`), and to return the newest solution with the fixed point iteration (see `KINSetReturnNewest`).

The installed `SUNDIALSConfig.cmake` file now supports the `COMPONENTS` option to `find_package`. The exported targets no longer have `IMPORTED_GLOBAL` set.

A bug was fixed in `SUNMatCopyOps` where the matrix-vector product setup function pointer was not copied.

A bug was fixed in the SPBCGS and SPTFQMR solvers for the case where a non-zero initial guess and a solution scaling vector are provided. This fix only impacts codes using SPBCGS or SPTFQMR as standalone solvers as all SUNDIALS packages utilize a zero initial guess.

A bug was fixed in the Picard iteration where the value of `KINSetMaxSetupCalls` would be ignored.

Changes in v5.7.0

A new NVECTOR implementation based on the SYCL abstraction layer has been added targeting Intel GPUs. At present the only SYCL compiler supported is the DPC++ (Intel oneAPI) compiler. See

Section 7.12 for more details. This module is considered experimental and is subject to major changes even in minor releases.

A new SUNMATRIX and SUNLINSOL implementation were added to interface with the MAGMA linear algebra library. Both the matrix and the linear solver support general dense linear systems as well as block diagonal linear systems, and both are targeted at GPUs (AMD or NVIDIA). See Section 9.13 for more details.

Changes in v5.6.1

Fixed a bug in the SUNDIALS CMake which caused an error if the CMAKE_CXX_STANDARD and SUNDIALS_RAJA_BACKENDS options were not provided.

Fixed some compiler warnings when using the IBM XL compilers.

Changes in v5.6.0

A new NVECTOR implementation based on the AMD ROCm HIP platform has been added. This vector can target NVIDIA or AMD GPUs. See 7.10 for more details. This module is considered experimental and is subject to change from version to version.

The RAJA NVECTOR implementation has been updated to support the HIP backend in addition to the CUDA backend. Users can choose the backend when configuring SUNDIALS by using the SUNDIALS_RAJA_BACKENDS CMake variable. This module remains experimental and is subject to change from version to version.

A new optional operation, `N_VGetDeviceArrayPointer`, was added to the N_Vector API. This operation is useful for N_Vectors that utilize dual memory spaces, e.g. the native SUNDIALS CUDA N_Vector.

The SUNMATRIX_CUSPARSE and SUNLINEARSOLVER_CUSOLVERSPP_BATCHQR implementations no longer require the SUNDIALS CUDA N_Vector. Instead, they require that the vector utilized provides the `N_VGetDeviceArrayPointer` operation, and that the pointer returned by `N_VGetDeviceArrayPointer` is a valid CUDA device pointer.

Changes in v5.5.0

Refactored the SUNDIALS build system. CMake 3.12.0 or newer is now required. Users will likely see deprecation warnings, but otherwise the changes should be fully backwards compatible for almost all users. SUNDIALS now exports CMake targets and installs a SUNDIALSConfig.cmake file.

Added support for SuperLU DIST 6.3.0 or newer.

Changes in v5.4.0

A new API, `SUNMemoryHelper`, was added to support **GPU users** who have complex memory management needs such as using memory pools. This is paired with new constructors for the `NVECTOR_CUDA` and `NVECTOR_RAJA` modules that accept a `SUNMemoryHelper` object. Refer to sections 6.1, 10.1, 7.9 and 7.11 for more information.

The `NVECTOR_RAJA` module has been updated to mirror the `NVECTOR_CUDA` module. Notably, the update adds managed memory support to the `NVECTOR_RAJA` module. Users of the module will need to update any calls to the `N_VMake_Raja` function because that signature was changed. This module remains experimental and is subject to change from version to version.

The `NVECTOR_TRILINOS` module has been updated to work with Trilinos 12.18+. This update changes the local ordinal type to always be an `int`.

Added support for CUDA v11.

Changes in v5.3.0

Fixed a bug in the iterative linear solver modules where an error is not returned if the `Atimes` function is `NULL` or, if preconditioning is enabled, the `PSolve` function is `NULL`.

Added the ability to control the CUDA kernel launch parameters for the `NVECTOR_CUDA` and `SUNMATRIX_CUSPARSE` modules. These modules remain experimental and are subject to change from version to version. In addition, the `NVECTOR_CUDA` kernels were rewritten to be more flexible. Most users should see equivalent performance or some improvement, but a select few may observe minor performance degradation with the default settings. Users are encouraged to contact the SUNDIALS team about any performance changes that they notice.

Added new capabilities for monitoring the solve phase in the `SUNNONLINSOL_NEWTON` and `SUNNONLINSOL_FIXEDPOINT` modules, and the SUNDIALS iterative linear solver modules. SUNDIALS must be built with the CMake option `SUNDIALS_BUILD_WITH_MONITORING` to use these capabilities.

Added the optional function `KINSetJacTimesVecSysFn` to specify an alternative system function for computing Jacobian-vector products with the internal difference quotient approximation.

Changes in v5.2.0

Fixed a build system bug related to the Fortran 2003 interfaces when using the IBM XL compiler. When building the Fortran 2003 interfaces with an XL compiler it is recommended to set `CMAKE_Fortran_COMPILER` to `f2003`, `xlf2003`, or `xlf2003_r`.

Fixed a linkage bug affecting Windows users that stemmed from `dllimport/dllexport` attributes missing on some SUNDIALS API functions.

Added a new `SUNMatrix` implementation, `SUNMATRIX_CUSPARSE`, that interfaces to the sparse matrix implementation from the NVIDIA cuSPARSE library. In addition, the `SUNLINSOL_CUSOLVER_BATCHQR` linear solver has been updated to use this matrix, therefore, users of this module will need to update their code. These modules are still considered to be experimental, thus they are subject to breaking changes even in minor releases.

Changes in v5.1.0

Fixed a build system bug related to finding LAPACK/BLAS.

Fixed a build system bug related to checking if the KLU library works.

Fixed a build system bug related to finding PETSc when using the CMake variables `PETSC_INCLUDES` and `PETSC_LIBRARIES` instead of `PETSC_DIR`.

Added a new build system option, `CUDA_ARCH`, that can be used to specify the CUDA architecture to compile for.

Added two utility functions, `SUNDIALSFileOpen` and `SUNDIALSFileClose` for creating/destroying file pointers that are useful when using the Fortran 2003 interfaces.

Added support for constant damping when using Anderson acceleration. See Chapter 2 and the description of the `KINSetDampingAA` function for more details.

Changes in v5.0.0

Build system changes

- Increased the minimum required CMake version to 3.5 for most SUNDIALS configurations, and 3.10 when CUDA or OpenMP with device offloading are enabled.
- The CMake option `BLAS_ENABLE` and the variable `BLAS_LIBRARIES` have been removed to simplify builds as SUNDIALS packages do not use BLAS directly. For third party libraries that require linking to BLAS, the path to the BLAS library should be included in the `_LIBRARIES` variable for the third party library *e.g.*, `SUPERLUDIST_LIBRARIES` when enabling `SuperLU_DIST`.
- Fixed a bug in the build system that prevented the `NVECTOR_PTHREADS` module from being built.

NVECTOR module changes

- Two new functions were added to aid in creating custom NVECTOR objects. The constructor `N_VNewEmpty` allocates an “empty” generic NVECTOR with the object’s content pointer and the function pointers in the operations structure initialized to `NULL`. When used in the constructor for custom objects this function will ease the introduction of any new optional operations to the NVECTOR API by ensuring only required operations need to be set. Additionally, the function `N_VCopyOps(w, v)` has been added to copy the operation function pointers between vector objects. When used in clone routines for custom vector objects these functions also will ease the introduction of any new optional operations to the NVECTOR API by ensuring all operations are copied when cloning objects. See §7.1.6 for more details.
- Two new NVECTOR implementations, `NVECTOR_MANYVECTOR` and `NVECTOR_MPIMANYVECTOR`, have been created to support flexible partitioning of solution data among different processing elements (e.g., CPU + GPU) or for multi-physics problems that couple distinct MPI-based simulations together. This implementation is accompanied by additions to user documentation and SUNDIALS examples. See §7.15 and §7.16 for more details.
- One new required vector operation and ten new optional vector operations have been added to the NVECTOR API. The new required operation, `N_VGetLength`, returns the global length of an `N_Vector`. The optional operations have been added to support the new `NVECTOR_MPIMANYVECTOR` implementation. The operation `N_VGetCommunicator` must be implemented by subvectors that are combined to create an `NVECTOR_MPIMANYVECTOR`, but is not used outside of this context. The remaining nine operations are optional local reduction operations intended to eliminate unnecessary latency when performing vector reduction operations (norms, etc.) on distributed memory systems. The optional local reduction vector operations are `N_VDotProdLocal`, `N_VMaxNormLocal`, `N_VMinLocal`, `N_VL1NormLocal`, `N_VWSqrSumLocal`, `N_VWSqrSumMaskLocal`, `N_VInvTestLocal`, `N_VConstrMaskLocal`, and `N_VMinQuotientLocal`. If an NVECTOR implementation defines any of the local operations as `NULL`, then the `NVECTOR_MPIMANYVECTOR` will call standard NVECTOR operations to complete the computation. See §7.1.4 for more details.
- An additional NVECTOR implementation, `NVECTOR_MPIPLUSX`, has been created to support the MPI+X paradigm where X is a type of on-node parallelism (e.g., OpenMP, CUDA). The implementation is accompanied by additions to user documentation and SUNDIALS examples. See §7.17 for more details.
- The `*MPICuda` and `*MPIRaja` functions have been removed from the `NVECTOR_CUDA` and `NVECTOR_RAJA` implementations respectively. Accordingly, the `nvector_mpicuda.h`, `nvector_mpiraja.h`, `libsundials_nvecmpicuda.lib`, and `libsundials_nvecmpicudaraja.lib` files have been removed. Users should use the `NVECTOR_MPIPLUSX` module coupled in conjunction with the `NVECTOR_CUDA` or `NVECTOR_RAJA` modules to replace the functionality. The necessary changes are minimal and should require few code modifications. See the programs in `examples/ida/mpicuda` and `examples/ida/mpiraja` for examples of how to use the `NVECTOR_MPIPLUSX` module with the `NVECTOR_CUDA` and `NVECTOR_RAJA` modules respectively.
- Fixed a memory leak in the `NVECTOR_PETSC` module clone function.
- Made performance improvements to the `NVECTOR_CUDA` module. Users who utilize a non-default stream should no longer see default stream synchronizations after memory transfers.
- Added a new constructor to the `NVECTOR_CUDA` module that allows a user to provide custom allocate and free functions for the vector data array and internal reduction buffer. See §7.9.1 for more details.
- Added new Fortran 2003 interfaces for most NVECTOR modules. See Chapter 7 for more details on how to use the interfaces.

- Added three new `NVECTOR` utility functions, `FN_VGetVecAtIndexVectorArray`, `FN_VSetVecAtIndexVectorArray`, and `FN_VNewVectorArray`, for working with `N_Vector` arrays when using the Fortran 2003 interfaces. See §7.1.6 for more details.

SUNMatrix module changes

- Two new functions were added to aid in creating custom `SUNMATRIX` objects. The constructor `SUNMatNewEmpty` allocates an “empty” generic `SUNMATRIX` with the object’s content pointer and the function pointers in the operations structure initialized to `NULL`. When used in the constructor for custom objects this function will ease the introduction of any new optional operations to the `SUNMATRIX` API by ensuring only required operations need to be set. Additionally, the function `SUNMatCopyOps(A, B)` has been added to copy the operation function pointers between matrix objects. When used in clone routines for custom matrix objects these functions also will ease the introduction of any new optional operations to the `SUNMATRIX` API by ensuring all operations are copied when cloning objects. See §8.1.2 for more details.
- A new operation, `SUNMatMatvecSetup`, was added to the `SUNMATRIX` API to perform any setup necessary for computing a matrix-vector product. This operation is useful for `SUNMATRIX` implementations which need to prepare the matrix itself, or communication structures before performing the matrix-vector product. Users who have implemented custom `SUNMATRIX` modules will need to at least update their code to set the corresponding `ops` structure member, `matvecsetup`, to `NULL`. See §8.1.1 for more details.
- The generic `SUNMATRIX` API now defines error codes to be returned by `SUNMATRIX` operations. Operations which return an integer flag indicating success/failure may return different values than previously. See §8.1.3 for more details.
- A new `SUNMATRIX` (and `SUNLINSOL`) implementation was added to facilitate the use of the `SuperLU_DIST` library with `SUNDIALS`. See §8.6 for more details.
- Added new Fortran 2003 interfaces for most `SUNMATRIX` modules. See Chapter 8 for more details on how to use the interfaces.

SUNLinearSolver module changes

- A new function was added to aid in creating custom `SUNLINSOL` objects. The constructor `SUNLinSolNewEmpty` allocates an “empty” generic `SUNLINSOL` with the object’s content pointer and the function pointers in the operations structure initialized to `NULL`. When used in the constructor for custom objects this function will ease the introduction of any new optional operations to the `SUNLINSOL` API by ensuring only required operations need to be set. See §9.3 for more details.
- The return type of the `SUNLINSOL` API function `SUNLinSolLastFlag` has changed from `long int` to `sunindextype` to be consistent with the type used to store row indices in dense and banded linear solver modules.
- Added a new optional operation to the `SUNLINSOL` API, `SUNLinSolGetID`, that returns a `SUNLinearSolver_ID` for identifying the linear solver module.
- The `SUNLINSOL` API has been updated to make the initialize and setup functions optional.
- A new `SUNLINSOL` (and `SUNMATRIX`) implementation was added to facilitate the use of the `SuperLU_DIST` library with `SUNDIALS`. See §9.10 for more details.
- Added a new `SUNLINSOL` implementation, `SUNLinearSolver_cuSolverSp_batchQR`, which leverages the `NVIDIA cuSOLVER` sparse batched QR method for efficiently solving block diagonal linear systems on `NVIDIA GPUs`. See §9.12 for more details.

- Added three new accessor functions to the SUNLINSOL_KLU module, `SUNLinSol_KLUGetSymbolic`, `SUNLinSol_KLUGetNumeric`, and `SUNLinSol_KLUGetCommon`, to provide user access to the underlying KLU solver structures. See §9.9.2 for more details.
- Added new Fortran 2003 interfaces for most SUNLINSOL modules. See Chapter 9 for more details on how to use the interfaces.

KINSOL changes

- Fixed a bug in the KINSOL linear solver interface where the auxiliary scalar `sJpnorm` was not computed when necessary with the Picard iteration and the auxiliary scalar `sFdotJp` was unnecessarily computed in some cases.
- The KINLS interface has been updated to only zero the Jacobian matrix before calling a user-supplied Jacobian evaluation function when the attached linear solver has type `SUNLINEARSOLVER_DIRECT`.
- Added a Fortran 2003 interface to KINSOL. See Chapter 5 for more details.

Changes in v5.0.0-dev.0

An additional NVECTOR implementation, `NVECTOR_MANYVECTOR`, was created to support flexible partitioning of solution data among different processing elements (e.g., CPU + GPU) or for multi-physics problems that couple distinct MPI-based simulations together (see Section 7.15 for more details). This implementation is accompanied by additions to user documentation and SUNDIALS examples.

Eleven new optional vector operations have been added to the NVECTOR API to support the new `NVECTOR_MANYVECTOR` implementation (see Chapter 7 for more details). Two of the operations, `N_VGetCommunicator` and `N_VGetLength`, must be implemented by subvectors that are combined to create an `NVECTOR_MANYVECTOR`, but are not used outside of this context. The remaining nine operations are optional local reduction operations intended to eliminate unnecessary latency when performing vector reduction operations (norms, etc.) on distributed memory systems. The optional local reduction vector operations are `N_VDotProdLocal`, `N_VMaxNormLocal`, `N_VMinLocal`, `N_VL1NormLocal`, `N_VWSqrSumLocal`, `N_VWSqrSumMaskLocal`, `N_VInvTestLocal`, `N_VConstrMaskLocal`, and `N_VMinQuotientLocal`. If an NVECTOR implementation defines any of the local operations as `NULL`, then the `NVECTOR_MANYVECTOR` will call standard NVECTOR operations to complete the computation.

A new `SUNMATRIX` and `SUNLINSOL` implementation was added to facilitate the use of the `SuperLU_DIST` library with SUNDIALS.

A new operation, `SUNMatMatvecSetup`, was added to the `SUNMATRIX` API. Users who have implemented custom `SUNMATRIX` modules will need to at least update their code to set the corresponding ops structure member, `matvecsetup`, to `NULL`.

The generic `SUNMATRIX` API now defines error codes to be returned by `SUNMATRIX` operations. Operations which return an integer flag indicating success/failure may return different values than previously.

Changes in v4.1.0

An additional NVECTOR implementation was added for the Tpetra vector from the Trilinos library to facilitate interoperability between SUNDIALS and Trilinos. This implementation is accompanied by additions to user documentation and SUNDIALS examples.

The `EXAMPLES_ENABLE_RAJA` CMake option has been removed. The option `EXAMPLES_ENABLE_CUDA` enables all examples that use CUDA including the RAJA examples with a CUDA back end (if the RAJA NVECTOR is enabled).

The implementation header file `kin_impl.h` is no longer installed. This means users who are directly manipulating the `KINMem` structure will need to update their code to use KINSOL's public API.

Python is no longer required to run `make test` and `make test_install`.

Changes in v4.0.2

Added information on how to contribute to SUNDIALS and a contributing agreement.

Moved definitions of DLS and SPILS backwards compatibility functions to a source file. The symbols are now included in the KINSOL library, `libsundials_kinsol`.

Changes in v4.0.1

No changes were made in this release.

Changes in v4.0.0

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. This includes the "DIRECT" and "ITERATIVE" types as well as the new "MATRIX_ITERATIVE" type. Details regarding how `KINLS` utilizes linear solvers of each type as well as discussion regarding intended use cases for user-supplied `SUNLINSOL` implementations are included in Chapter 9. All KINSOL example programs and the standalone linear solver examples have been updated to use the unified linear solver interface.

The unified interface for the new `KINLS` module is very similar to the previous `KINDLS` and `KINSPILS` interfaces. 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. Additionally, we 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 follow the naming convention `SUNLinSol_*` where `*` is the name of the linear solver. The new names are `SUNLinSol_Band`, `SUNLinSol_Dense`, `SUNLinSol_KLU`, `SUNLinSol_LapackBand`, `SUNLinSol_LapackDense`, `SUNLinSol_PCG`, `SUNLinSol_SPBCGS`, `SUNLinSol_SPGMR`, `SUNLinSol_SPGMR`, `SUNLinSol_SPTFQMR`, and `SUNLinSol_SuperLUMT`. 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. All KINSOL example programs and the standalone linear solver examples have been updated to use the new naming convention.

The `SUNBandMatrix` constructor has been simplified to remove the storage upper bandwidth argument.

Three fused vector operations and seven vector array operations have been added to the `NVECTOR` API. These *optional* operations are disabled by default and may be activated by calling vector specific routines after creating an `NVECTOR` (see Chapter 7 for more details). The new 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 fused operations are `N_VLinearCombination`, `N_VScaleAddMulti`, and `N_VDotProdMulti` and the vector array operations are `N_VLinearCombinationVectorArray`, `N_VScaleVectorArray`, `N_VConstVectorArray`, `N_VWrmsNormVectorArray`, `N_VWrmsNormMaskVectorArray`, `N_VScaleAddMultiVectorArray`, and `N_VLinearCombinationVectorArray`. If an `NVECTOR` implementation defines any of these operations as `NULL`, then standard `NVECTOR` operations will automatically be called as necessary to complete the computation.

Multiple updates to `NVECTOR_CUDA` were made:

- Changed `N_VGetLength_Cuda` to return the global vector length instead of the local vector length.
- Added `N_VGetLocalLength_Cuda` to return the local vector length.
- Added `N_VGetMPIComm_Cuda` to return the MPI communicator used.
- Removed the accessor functions in the namespace `suncudavec`.
- Changed the `N_VMake_Cuda` function to take a host data pointer and a device data pointer instead of an `N_VectorContent_Cuda` object.
- Added the ability to set the `cudaStream_t` used for execution of the `NVECTOR_CUDA` kernels. See the function `N_VSetCudaStreams_Cuda`.
- Added `N_VNewManaged_Cuda`, `N_VMakeManaged_Cuda`, and `N_VIsManagedMemory_Cuda` functions to accommodate using managed memory with the `NVECTOR_CUDA`.

Multiple changes to `NVECTOR_RAJA` were made:

- Changed `N_VGetLength_Raja` to return the global vector length instead of the local vector length.
- Added `N_VGetLocalLength_Raja` to return the local vector length.
- Added `N_VGetMPIComm_Raja` to return the MPI communicator used.
- Removed the accessor functions in the namespace `suncudavec`.

A new `NVECTOR` implementation for leveraging OpenMP 4.5+ device offloading has been added, `NVECTOR_OPENMPDEV`. See §7.13 for more details.

Changes in v3.2.1

The changes in this minor release include the following:

- Fixed a bug in the `CUDA NVECTOR` where the `N_VInvTest` operation could write beyond the allocated vector data.
- Fixed library installation path for multiarch systems. This fix changes the default library installation path to `CMAKE_INSTALL_PREFIX/CMAKE_INSTALL_LIBDIR` from `CMAKE_INSTALL_PREFIX/lib`. `CMAKE_INSTALL_LIBDIR` is automatically set, but is available as a CMake option that can be modified.

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., `ENABLE_LAPACK` 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_F77_FUNC_CASE` and `SUNDIALS_F77_FUNC_UNDERSCORES` 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 `sunmatrix_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` `Dls` and `SlS` matrix structures in a single object-oriented API.
- Added example problems demonstrating use of generic `SUNMATRIX` modules.
- Added generic `SUNLinearSolver` module with eleven provided implementations: `SUNDIALS` native dense, `SUNDIALS` native banded, `LAPACK` dense, `LAPACK` band, `KLU`, `SuperLU_MT`, `SPGMR`, `SPBCGS`, `SPTFQMR`, `SPFGMR`, and `PCG`. These replicate previous `SUNDIALS` generic linear solvers in a single object-oriented API.
- Added example problems demonstrating use of generic `SUNLINEARSOLVER` modules.
- Expanded package-provided direct linear solver (`Dls`) interfaces and scaled, preconditioned, iterative linear solver (`Spils`) interfaces to utilize generic `SUNMATRIX` and `SUNLINEARSOLVER` objects.
- Removed package-specific, linear solver-specific, solver modules (e.g. `CVDENSE`, `KINBAND`, `IDAKLU`, `ARKSPGMR`) since their functionality is entirely replicated by the generic `Dls/Spils` interfaces and `SUNLINEARSOLVER/SUNMATRIX` modules. The exception is `CVDIAG`, a diagonal approximate Jacobian solver available to `CVODE` and `CVODES`.
- Converted all `SUNDIALS` example problems to utilize new generic `SUNMATRIX` and `SUNLINEARSOLVER` objects, along with updated `Dls` and `Spils` linear solver interfaces.
- Added `Spils` interface routines to `ARKode`, `CVODE`, `CVODES`, `IDA` and `IDAS` to allow specification of a user-provided "JTSetup" routine. This change supports users who wish to set up data structures for the user-provided Jacobian-times-vector ("JTimes") routine, and where the cost of one `JTSetup` setup per Newton iteration can be amortized between multiple `JTimes` calls.

Two additional NVECTOR implementations were added – one for CUDA and one for RAJA vectors. These vectors are supplied to provide very basic support for running on GPU architectures. Users are advised that these vectors both move all data to the GPU device upon construction, and speedup will only be realized if the user also conducts the right-hand-side function evaluation on the device. In addition, these vectors assume the problem fits on one GPU. Further information about RAJA, users are referred to 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 `boolean` type values `TRUE` and `FALSE` have been changed to `SUNTRUE` and `SUNFALSE` respectively.

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

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

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

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

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

A bug fix was done to correct the `fcmix` name translation for `FKIN_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, `NVGetVectorID`, 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 `init` function. This ensures that these solver counters are initialized upon linear solver instantiation as well as at the beginning of the problem solution.

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

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

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

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

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

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

Changes in v2.8.0

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

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

Otherwise, only relatively minor modifications were made to KINSOL:

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

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

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

In `KINLapackBand`, the line `smu = MIN(N-1,mu+m1)` was changed to `smu = mu + m1` 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`, `SUNRsqr`, `SUNRexp`, `SRpowerI`, and `SUNRpowerR`, respectively. These names occur in both the solver and in various example programs.

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

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

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

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

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

Changes in v2.7.0

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

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

Changes in v2.6.0

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

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

Changes in v2.5.0

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

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

Changes in v2.4.0

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

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

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

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

Changes in v2.3.0

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

Changes in v2.2.1

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

Changes in v2.2.0

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

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

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

1.3 Reading this User Guide

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

There are different possible levels of usage of KINSOL. The most casual user, with a small nonlinear system, can get by with reading all of Chapter 2, then Chapter 4 through §4.5.3 only, and looking at examples in [15]. 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 7), or even (d) supply a different linear solver module (§3.2 and Chapter 9).

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 5.1.5, we describe FKINSOL, an interface module for the use of KINSOL with FORTRAN applications.
- Chapter 7 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 8 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 (§8.3), a banded implementation (§8.4) and a sparse implementation (§8.5).
- Chapter 9 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

All SUNDIALS packages are released open source, under the BSD 3-Clause license. The only requirements of the license are preservation of copyright and a standard disclaimer of liability. The full text of the license and an additional notice are provided below and may also be found in the LICENSE and NOTICE files provided with all SUNDIALS packages.



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 BSD 3-Clause License

Copyright (c) 2002-2021, Lawrence Livermore National Security and Southern Methodist University. All rights reserved.

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

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

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

* Neither the name of the copyright holder 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 THE COPYRIGHT HOLDER 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.

1.4.2 Additional Notice

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

This work 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.

1.4.3 SUNDIALS Release Numbers

LLNL-CODE-667205 (ARKODE)
UCRL-CODE-155951 (CVOICE)
UCRL-CODE-155950 (CVOICES)
UCRL-CODE-155952 (IDA)
UCRL-CODE-237203 (IDAS)
LLNL-CODE-665877 (KINSOL)

Chapter 2

Mathematical Considerations

KINSOL solves nonlinear algebraic systems in real N -space.

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

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

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

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

given an initial guess u_0 .

Basic Newton iteration

Depending on the linear solver used, KINSOL can employ either an Inexact Newton method [11, 13, 17, 19, 30], or a Modified Newton method. At the highest level, KINSOL implements the following iteration scheme:

1. Set $u_0 =$ an initial guess
2. For $n = 0, 1, 2, \dots$ 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 δ_n , is added to u_n to produce a new iterate, u_{n+1} . A test for convergence is made before the iteration continues.

Newton method variants

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

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

- band direct solvers, using either an internal implementation or a BLAS/LAPACK implementation (serial or threaded vector modules only),
- sparse direct solver interfaces, using either the KLU sparse solver library [16, 3], or the thread-enabled SuperLU_MT sparse solver library [31, 18, 9] (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 below¹). Note that the dense, band, and sparse direct linear solvers can only be used with the serial and threaded vector representations.

When using an iterative linear solver, the linear system in (2a) 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),
- `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 `mbset`, the number of nonlinear iterations after which a Jacobian information update is enforced.

¹KINSOL 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:

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

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

Globalization strategy

Two methods of applying a computed step δ_n to the previously computed solution vector are implemented. The first and simplest is the standard Newton strategy which applies step 2(b) as above with λ always set to 1. The other method is a global strategy, which attempts to use the direction implied by δ_n in the most efficient way for furthering convergence of the nonlinear problem. This technique is implemented in the second strategy, called Linesearch. This option employs both the α and β conditions of the Goldstein-Armijo linesearch given in [19] for step 2(b), where λ is chosen to guarantee a sufficient decrease in F relative to the step length as well as a minimum step length relative to the initial rate of decrease of F . One property of the algorithm is that the full Newton step tends to be taken close to the solution.

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

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

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

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

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

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

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

For more details, the reader is referred to [19].

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 δ_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 `mbsetsub=5` nonlinear iterations have passed since the last update and

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

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

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

with ρ defined as

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

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

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

Stopping criteria for iterative linear solvers

When using an Inexact Newton method (i.e. when an iterative linear solver is used), the convergence of the overall nonlinear solver is intimately coupled with the accuracy with which the linear solver in 2(a) above is solved. KINSOL provides three options for stopping criteria for the linear system solver, including the two algorithms of Eisenstat and Walker [21]. More precisely, the Krylov iteration must pass a stopping test

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

where η_n is one of:

Eisenstat and Walker Choice 1

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

Eisenstat and Walker Choice 2

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

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

Constant η

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

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 η does not decrease too quickly [21].

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

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

The increments σ_j are given by

$$\sigma_j = \sqrt{U} \max\{|u^j|, 1/D_u^j\}. \quad (2.7)$$

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

We note that with 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

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

where u is the current approximation to a root of (2.1), and σ is a scalar. The choice of σ is taken from [13] and is given by

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

where u_{typ} is a vector of typical values for the absolute values of the solution (and can be taken to be inverses of the scale factors given for u as described below). This formula is suitable for *scaled* vectors u and v , and so is applied to $D_u u$ and $D_u v$. The parameter U above can (optionally) be replaced by a user-specified value, `relfunc`. Convergence of the Newton method is maintained as long as the value of σ remains appropriately small, as shown in [11].

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 $n = 0, 1, 2, \dots$ until convergence do:
 - (a) Set $u_{n+1} = (1 - \beta)u_n + \beta G(u_n)$.
 - (b) Test for convergence.

Here, u_n is the n -th iterate to u . At each stage in the iteration process, the function G is applied to the current iterate with the damping parameter β 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 $n = 0, 1, 2, \dots$ until convergence do:
 - (a) Set $u_{n+1} = (1 - \beta)u_n + \beta G(u_n)$ where $G(u_n) \equiv u_n - L^{-1}F(u_n)$.
 - (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 with the damping parameter β 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 [35].

Anderson Acceleration

The Picard and fixed point methods can be significantly accelerated using Anderson's method [10, 41, 22, 34]. 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 $n = 1, 2, \dots$ until convergence do:
 - (a) Set $m_n = \min\{m, n\}$
 - (b) Set $F_n = (f_{n-m_n}, \dots, f_n)$, where $f_i = G(u_i) - u_i$
 - (c) Determine $\alpha^{(n)} = (\alpha_0^{(n)}, \dots, \alpha_{m_n}^{(n)})$ that solves $\min_{\alpha} \|F_n \alpha^T\|_2$ such that $\sum_{i=0}^{m_n} \alpha_i = 1$
 - (d) Set $u_{n+1} = \beta \sum_{i=0}^{m_n} \alpha_i^{(n)} G(u_{n-m_n+i}) + (1 - \beta) \sum_{i=0}^{m_n} \alpha_i^{(n)} u_{n-m_n+i}$
 - (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 $n = 1, 2, \dots$ until convergence do:
 - (a) Set $m_n = \min\{m, n\}$
 - (b) Set $\Delta F_n = (\Delta f_{n-m_n}, \dots, \Delta f_{n-1})$, where $\Delta f_i = f_{i+1} - f_i$ and $f_i = G(u_i) - u_i$
 - (c) Determine $\gamma^{(n)} = (\gamma_0^{(n)}, \dots, \gamma_{m_n-1}^{(n)})$ that solves $\min_{\gamma} \|f_n - \Delta F_n \gamma^T\|_2$
 - (d) Set $u_{n+1} = G(u_n) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta g_{n-m_n+i} - (1 - \beta)(f(u_n) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta f_{n-m_n+i})$ 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 $R_n \gamma = Q_n^T f_n$. By default the damping is disabled i.e., $\beta = 1.0$.

The Anderson acceleration implementation includes an option to delay the start of acceleration until after a given number of initial fixed-point or Picard iterations have been completed. This delay can be beneficial when the underlying method has strong global convergence properties as the initial iterations may help bring the iterates closer to a solution before starting the acceleration.

Fixed-point - Anderson Acceleration Stopping Criterion

The default stopping criterion is

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

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

Picard - Anderson Acceleration Stopping Criterion

The default stopping criterion is

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

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

Chapter 3

Code Organization

3.1 SUNDIALS organization

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

The various solvers of this family share many subordinate modules. For this reason, it is organized as a family, with a directory structure that exploits that sharing (see Figures 3.1 and 3.2). The

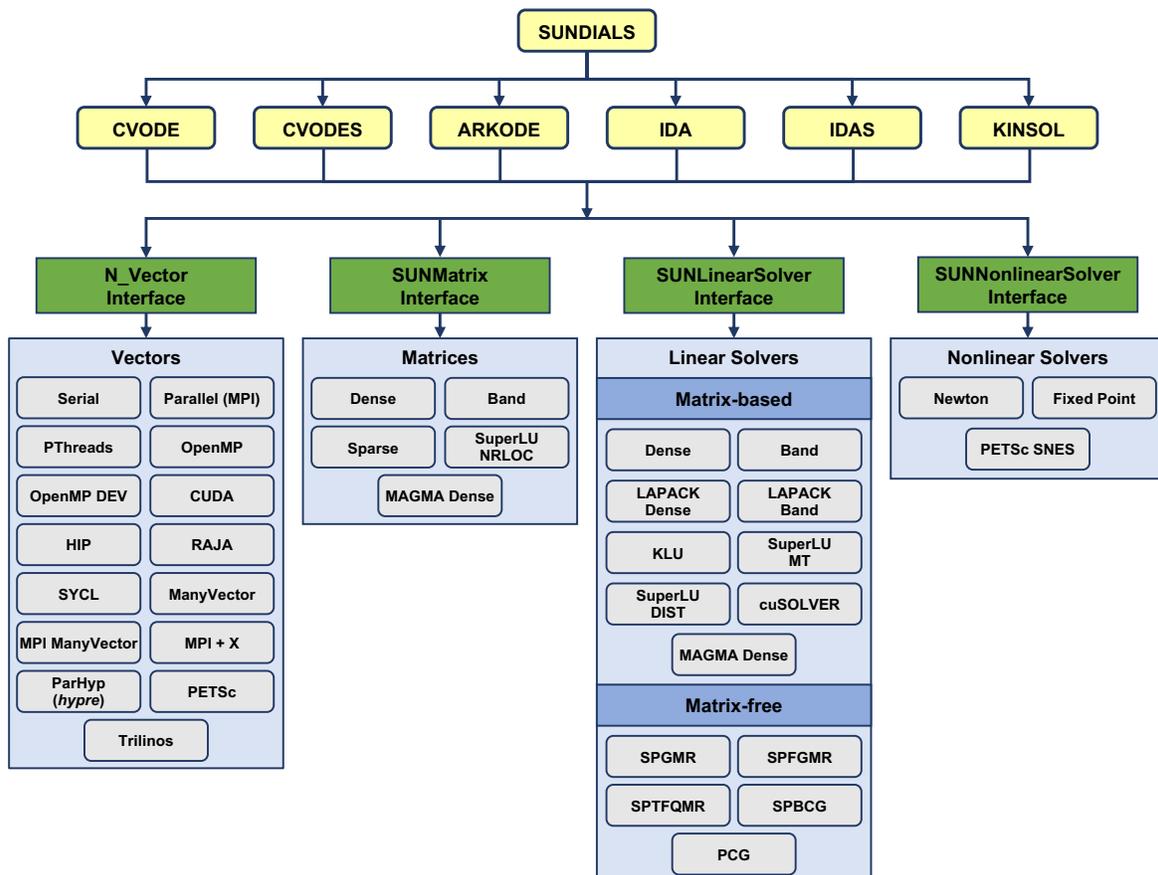


Figure 3.1: High-level diagram of the SUNDIALS suite.

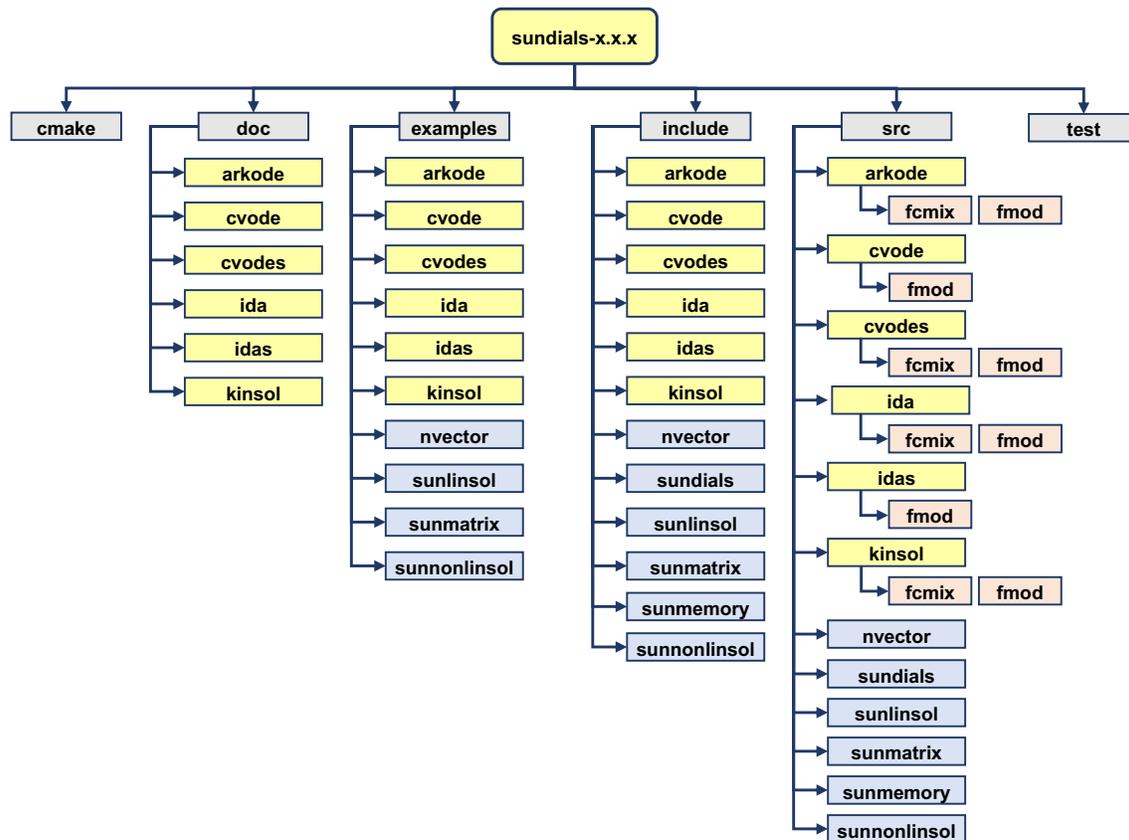


Figure 3.2: Directory structure of the SUNDIALS source tree.

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 stiff, nonstiff, mixed stiff-nonstiff, and multirate ODE systems $Mdy/dt = f_1(t, y) + f_2(t, y)$ based on Runge-Kutta methods;
- IDA, a solver for differential-algebraic systems $F(t, y, \dot{y}) = 0$ based on BDF methods;
- IDAS, a solver for differential-algebraic systems with sensitivity analysis capabilities;
- KINSOL, a solver for nonlinear algebraic systems $F(u) = 0$.

Note for modules that provide interfaces to third-party libraries (i.e., LAPACK, KLU, SUPERLUMT, SuperLU_DIST, *hypre*, PETSc, Trilinos, and RAJA) users will need to download and compile those packages independently.

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

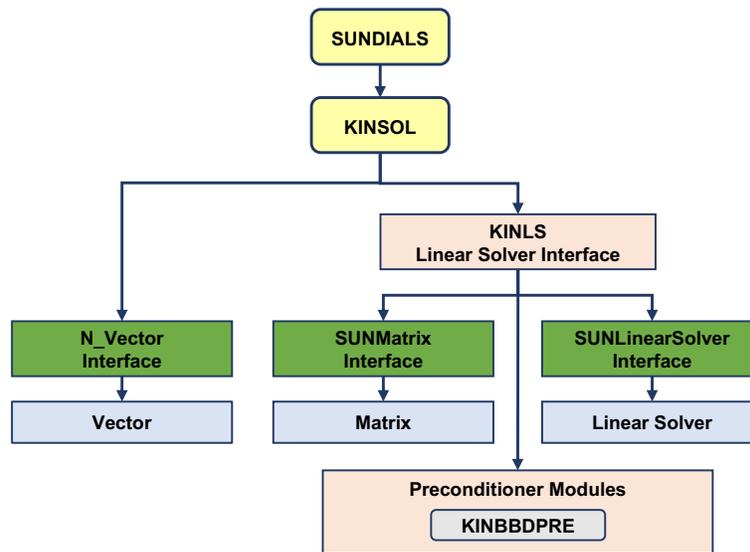


Figure 3.3: Overall structure diagram of the KINSOL package. Modules specific to KINSOL begin with “KIN” (KINLS and KINBBDPRE), all other items correspond to generic SUNDIALS vector, matrix, and solver modules (see Figure 3.1).

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 9). These solvers may utilize a SUNMATRIX object (see Chapter 8) 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 [12, 14], together with the example and demonstration programs included with KINSOL, offer considerable assistance in building preconditioners.

KINSOL’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 KINSOL module to each of the associated functions is fixed, thus allowing the central module to be completely independent of the linear system method.

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

§4.7.

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

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 [15] 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 5.1.5, which describes the FORTRAN/C interface module.

The user should be aware that not all SUNLINSOL and SUNMATRIX modules are compatible with all NVECTOR implementations. Details on compatability are given in the documentation for each SUNMATRIX module (Chapter 8) and each SUNLINSOL module (Chapter 9). 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 8 and 9 to verify compatability between these modules. In addition to that documentation, we note that the preconditioner module KINBBDPRE can only be used with NVECTOR_PARALLEL. It is not recommended to use a threaded vector module with SuperLU_MT unless it is the NVECTOR_OPENMP module, and SuperLU_MT is also compiled with OpenMP.

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

4.1 Access to library and header files

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

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

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

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

- *incdir/include/kinsol*
- *incdir/include/sundials*
- *incdir/include/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,

```
#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.

Additionally, SUNDIALS defines several macros for common mathematical functions *e.g.*, `fabs`, `sqrt`, `exp`, etc. in `sundials_math.h`. The macros are prefixed with `SUNR` and expand to the appropriate C function based on the `realtype`. For example, the macro `SUNRabs` expands to the C function `fabs` when `realtype` is `double`, `fabsf` when `realtype` is `float`, and `fabsl` when `realtype` is `long double`.

A user program which uses the type `realtype`, the `RCONST` macro, and the `SUNR` mathematical function macros is precision-independent except for any calls to precision-specific library functions. Our example programs use `realtype`, `RCONST`, and the `SUNR` macros. 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`) and call the appropriate math library functions directly. Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use `realtype`, `RCONST`, or the `SUNR` macros so long as the SUNDIALS libraries use the correct precision (for details see §A.1.2).

4.2.2 Integer types used for indexing

The type `sunindextype` is used for indexing array entries in SUNDIALS modules (*e.g.*, vectors lengths and matrix sizes) as well as for storing the total problem size. During configuration `sunindextype` may be selected to be either a 32- or 64-bit *signed* integer with the default being 64-bit. See §A.1.2 for the configuration option to select the desired size of `sunindextype`. When using a 32-bit integer the total problem size is limited to $2^{31} - 1$ and with 64-bit integers the limit is $2^{63} - 1$. For users with problem sizes that exceed the 64-bit limit an advanced configuration option is available to specify the type used for `sunindextype`.

A user program which uses `sunindextype` to handle 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 compatible type (*e.g.*, `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 `realttype`, `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 7 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 SUNDIALS-provided 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 module, `SUNLINSOL_LAPACKDENSE`;
 - `sunlinsol/sunlinsol_lapackband.h`, which is used with the LAPACK package banded linear solver module, `SUNLINSOL_LAPACKBAND`;
 - `sunlinsol/sunlinsol_klu.h`, which is used with the KLU sparse linear solver module, `SUNLINSOL_KLU`;
 - `sunlinsol/sunlinsol_superlumt.h`, which is used with the SUPERLUMT sparse linear solver 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_spgmrs.h`, which is used with the scaled, preconditioned Bi-CGStab Krylov linear solver module, `SUNLINSOL_SPGMRS`;
 - `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_kry_p` example (see [15]), 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 7, 8, and 9 for the specific name of the function to be called or macro to be referenced.

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

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

2. Set problem dimensions etc.

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

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

3. Set vector with initial guess

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

For native SUNDIALS vector implementations, use a call of the form `u = N_VMake_***(..., udata)` if the `realttype` array `udata` containing the initial values of `u` already exists. Otherwise, create a new vector by making a call of the form `u = N_VNew_***(...)`, and then set its elements by accessing the underlying data with a call of the form `ydata = N_VGetArrayPointer(u)`. See §7.3-7.6 for details.

For the `hypr` and PETSc vector wrappers, first create and initialize the underlying vector and then create an NVECTOR wrapper with a call of the form `u = N_VMake_***(uvec)`, where `uvec` is a `hypr` or PETSc vector. Note that calls like `N_VNew_***(...)` and `N_VGetArrayPointer(...)` are not available for these vector wrappers. See §7.7 and §7.8 for details.

4. Create KINSOL object

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

5. Allocate internal memory

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

value. See §4.5.1 for details.

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

```
SUNMatrix J = SUNBandMatrix(...);
```

or

```
SUNMatrix J = SUNDenseMatrix(...);
```

or

```
SUNMatrix J = 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

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

where * can be replaced with “Dense”, “SPGMR”, or other options, as discussed in §4.5.2 and Chapter 9.

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 9 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):

```
ier = 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. Deallocate memory for solution vector

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

```
N_VDestroy(u);
```

14. Free solver memory

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

15. Free linear solver and matrix memory

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

16. Finalize MPI, if used

Call `MPI_Finalize()` to terminate MPI.

SUNDIALS provides some linear solvers only as a means for users to get problems running and not as highly efficient solvers. For example, if solving a dense system, we suggest using the LAPACK solvers if the size of the linear system is $> 50,000$. (Thanks to A. Nicolai for his testing and recommendation.) Table 4.1 shows the linear solver interfaces available as SUNLINSOL modules and the vector implementations required for use. As an example, one cannot use the dense direct solver interfaces with the MPI-based vector implementation. However, as discussed in Chapter 9 the SUNDIALS packages operate on generic SUNLINSOL objects, allowing a user to develop their own solvers should they so desire.

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

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

4.5 User-callable functions

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

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

4.5.1 KINSOL initialization and deallocation functions

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

first two calls.

KINCreate

Call `kin_mem = KINCreate();`

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

Arguments This function has no arguments.

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

KINInit

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

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.)
`tpl` (`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, on both the left and the right, 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_SPGMR`, `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 8 and 9.

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	<code>flag = KINSetLinearSolver(kin_mem, LS, J);</code>
Description	The function <code>KINSetLinearSolver</code> attaches a generic <code>SUNLINSOL</code> object <code>LS</code> and corresponding template Jacobian <code>SUNMATRIX</code> object <code>J</code> (if applicable) to KINSOL, initializing the KINLS linear solver interface.
Arguments	<p><code>kin_mem</code> (void *) pointer to the KINSOL memory block.</p> <p><code>LS</code> (<code>SUNLinearSolver</code>) <code>SUNLINSOL</code> object to use for solving Newton linear systems.</p> <p><code>J</code> (<code>SUNMatrix</code>) <code>SUNMATRIX</code> object for used as a template for the Jacobian (or <code>NULL</code> if not applicable).</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>KINLS_SUCCESS</code> The KINLS initialization was successful.</p> <p><code>KINLS_MEM_NULL</code> The <code>kin_mem</code> pointer is <code>NULL</code>.</p> <p><code>KINLS_ILL_INPUT</code> The KINLS interface is not compatible with the <code>LS</code> or <code>J</code> input objects or is incompatible with the current <code>NVECTOR</code> module.</p> <p><code>KINLS_SUNLS_FAIL</code> A call to the <code>LS</code> object failed.</p> <p><code>KINLS_MEM_FAIL</code> A memory allocation request failed.</p>
Notes	<p>If <code>LS</code> is a matrix-based linear solver, then the template Jacobian matrix <code>J</code> will be used in the solve process, so if additional storage is required within the <code>SUNMATRIX</code> 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 <code>SUNMATRIX</code> type in Chapter 8 for further information).</p> <p>The previous routines <code>KINDlsSetLinearSolver</code> and <code>KINSpilsSetLinearSolver</code> 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.</p>

4.5.3 KINSOL solver function

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

KINSol

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

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

Arguments

- `kin_mem` (void *) pointer to the KINSOL memory block.
- `u` (`N_Vector`) vector set to initial guess by user before calling `KINSol`, but which upon return contains an approximate solution of the nonlinear system $F(u) = 0$.
- `strategy` (int) strategy used to solve the nonlinear system. It must be of the following:
 - `KIN_NONE` basic Newton iteration
 - `KIN_LINESEARCH` Newton with globalization
 - `KIN_FP` fixed-point iteration with Anderson Acceleration (no linear solver needed)
 - `KIN_PICARD` Picard iteration with Anderson Acceleration (uses a linear solver)
- `u_scale` (`N_Vector`) vector containing diagonal elements of scaling matrix D_u for vector `u` chosen so that the components of $D_u \cdot u$ (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 $u = u_0$ satisfied the system $F(u) = 0$ within the tolerances specified (the scaled norm of $F(u_0)$ is less than $0.01 \cdot \text{fnormtol}$).
- `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 `KINSetScaledStepTo1` in §4.5.4 to change `scsteptol` from its default value).
- `KIN_MEM_NULL`
 - The `KINSOL` memory block pointer was `NULL`.
- `KIN_ILL_INPUT`
 - An input parameter was invalid.
- `KIN_NO_MALLOC`
 - The `KINSOL` memory was not allocated by a call to `KINCreate`.
- `KIN_MEM_FAIL`
 - A memory allocation failed.
- `KIN_LINESEARCH_NONCONV`
 - The line search algorithm was unable to find an iterate sufficiently distinct from the current iterate, or could not find an iterate satisfying the sufficient decrease condition.

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

KIN_MAXITER_REACHED

The maximum number of nonlinear iterations has been reached.

KIN_MXNEWT_5X_EXCEEDED

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

KIN_LINESEARCH_BCFAIL

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

KIN_LINSOLV_NO_RECOVERY

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

KIN_LINIT_FAIL

The KINLS initialization routine (`linit`) encountered an error.

KIN_LSETUP_FAIL

The KINLS setup routine (`lsetup`) encountered an error; e.g., the user-supplied routine `pset` (used to set up the preconditioner data) encountered an unrecoverable error.

KIN_LSOLVE_FAIL

The KINLS solve routine (`lsolve`) encountered an error; e.g., the user-supplied routine `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_scale` and `f_scale` should be strictly positive.

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

4.5.4 Optional input functions

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

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

Table 4.2: Optional inputs for KINSOL and KINLS

Optional input	Function name	Default
KINSOL main solver		
Error handler function	KINSetErrHandlerFn	internal fn.
Pointer to an error file	KINSetErrFile	stderr
Info handler function	KINSetInfoHandlerFn	internal fn.
Pointer to an info file	KINSetInfoFile	stdout
Data for problem-defining function	KINSetUserData	NULL
Verbosity level of output	KINSetPrintLevel	0
Max. number of nonlinear iterations	KINSetNumMaxIters	200
No initial matrix setup	KINSetNoInitSetup	SUNFALSE
No residual monitoring*	KINSetNoResMon	SUNFALSE
Max. iterations without matrix setup	KINSetMaxSetupCalls	10
Max. iterations without residual check*	KINSetMaxSubSetupCalls	5
Form of η coefficient	KINSetEtaForm	KIN_ETACHOICE1
Constant value of η	KINSetEtaConstValue	0.1
Values of γ and α	KINSetEtaParams	0.9 and 2.0
Values of ω_{min} and ω_{max} *	KINSetResMonParams	0.00001 and 0.9
Constant value of ω^*	KINSetResMonConstValue	0.9
Lower bound on ϵ	KINSetNoMinEps	SUNFALSE
Max. scaled length of Newton step	KINSetMaxNewtonStep	$1000\ D_u u_0\ _2$
Max. number of β -condition failures	KINSetMaxBetaFails	10
Rel. error for D.Q. Jv	KINSetRelErrFunc	$\sqrt{\text{around}}$
Function-norm stopping tolerance	KINSetFuncNormTol	$\text{around}^{1/3}$
Scaled-step stopping tolerance	KINSetScaledSteptol	$\text{around}^{2/3}$
Inequality constraints on solution	KINSetConstraints	NULL
Nonlinear system function	KINSetSysFunc	none
Return the newest fixed point iteration	KINSetReturnNewest	SUNFALSE
Fixed point/Picard damping parameter	KINSetDamping	1.0
Anderson Acceleration subspace size	KINSetMAA	0
Anderson Acceleration damping parameter	KINSetDampingAA	1.0
Anderson Acceleration delay	KINSetDelayAA	0
KINLS linear solver interface		
Jacobian function	KINSetJacFn	DQ
Preconditioner functions and data	KINSetPreconditioner	NULL, NULL, NULL
Jacobian-times-vector function and data	KINSetJacTimesVecFn	internal DQ, NULL
Jacobian-times-vector system function	KINSetJacTimesVecSysFn	NULL

4.5.4.1 Main solver optional input functions

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

`KINSetErrFile`

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

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

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

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

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

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

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



`KINSetErrHandlerFn`

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

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

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

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

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

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

`KINSetInfoFile`

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

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

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

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

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

KINSetInfoHandlerFn

- Call `flag = KINSetInfoHandlerFn(kin_mem, ihfun, ih_data);`
- Description The function `KINSetInfoHandlerFn` specifies the optional user-defined function to be used in handling informative (non-error) messages.
- Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`ihfun` (`KINInfoHandlerFn`) is the user's C information handler function (see §4.6.3).
`ih_data` (`void *`) pointer to user data passed to `ihfun` every time it is called.
- Return value The return value `flag` (of type `int`) is one of:
`KIN_SUCCESS` The function `ihfun` and data pointer `ih_data` have been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
- Notes The default internal information handler function directs informative (non-error) messages to the file specified by the file pointer `infofp` (see `KINSetInfoFile` above).

KINSetPrintLevel

- Call `flag = KINSetPrintLevel(kin_mem, printf1);`
- Description The function `KINSetPrintLevel` specifies the level of verbosity of the output.
- Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`printf1` (`int`) flag indicating the level of verbosity. Must be one of:
- 0 no information displayed.
 - 1 for each nonlinear iteration display the following information: the scaled Euclidean ℓ_2 norm of the system function evaluated at the current iterate, the scaled norm of the Newton step (only if using `KIN_NONE`), and the number of function evaluations performed so far.
 - 2 display level 1 output and the following values for each iteration:
 $\|F(u)\|_{D_F}$ (only for `KIN_NONE`).
 $\|F(u)\|_{D_F, \infty}$ (for `KIN_NONE` and `KIN_LINESEARCH`).
 - 3 display level 2 output plus additional values used by the global strategy (only if using `KIN_LINESEARCH`), and statistical information for 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 `printf1` had an illegal value.
- Notes The default value for `printf1` 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.



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`.

The value of `msbset` should be a multiple of `msbsetsub` (see `KINSetMaxSubSetupCalls`).

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`.

The value of `msbset` (see `KINSetMaxSetupCalls`) should be a multiple of `msbsetsub`.

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 η coefficient used in the calculation of the linear solver convergence tolerance.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`etachoice` (`int`) flag indicating the method for computing η . The value must be one of `KIN_ETACHOICE1`, `KIN_ETACHOICE2`, or `KIN_ETACONSTANT` (see Chapter 2 for details).

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

`KIN_SUCCESS` The optional value has been successfully set.

`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT` The argument `etachoice` had an illegal value.

Notes The default value for `etachoice` is `KIN_ETACHOICE1`.
 When using either `KIN_ETACHOICE1` or `KIN_ETACHOICE2` the safeguard

$$\eta_n = \max(\eta_n, \eta_{\text{safe}})$$

is applied when $\eta_{\text{safe}} > 0.1$. For `KIN_ETACHOICE1`

$$\eta_{\text{safe}} = \eta_{n-1}^{\frac{1+\sqrt{5}}{2}}$$

and for `KIN_ETACHOICE2`

$$\eta_{\text{safe}} = \gamma \eta_{n-1}^\alpha$$

where γ and α can be set with `KINSetEtaParams`.
 The following safeguards are always applied when using either `KIN_ETACHOICE1` or `KIN_ETACHOICE2` so that $\eta_{\min} \leq \eta_n \leq \eta_{\max}$:

$$\eta_n = \max(\eta_n, \eta_{\min})$$

$$\eta_n = \min(\eta_n, \eta_{\max})$$

where $\eta_{\min} = 10^{-4}$ and $\eta_{\max} = 0.9$.

KINSetEtaConstValue

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

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

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

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

Notes The default value for `eta` is 0.1. The legal values are $0.0 < \text{eta} \leq 1.0$.

KINSetEtaParams

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

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

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

Return value The return value `flag` (of type `int`) is one of:
`KIN_SUCCESS` The optional values have been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT` One of the arguments `egamma` or `ealpha` had an illegal value.

Notes The default values for `egamma` and `ealpha` are 0.9 and 2.0, respectively.
 The legal values are $0.0 < \text{egamma} \leq 1.0$ and $1.0 < \text{ealpha} \leq 2.0$.

KINSetResMonConstValue

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

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

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

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

Notes The default value for `omegaconst` is 0.9. The legal values are $0.0 < \text{omegaconst} < 1.0$.

KINSetResMonParams

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

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

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

Return value The return value `flag` (of type `int`) is one of:
`KIN_SUCCESS` The optional values have been successfully set.
`KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT` One of the arguments `omegamin` or `omegamax` had an illegal value.

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

KINSetNoMinEps

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

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

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`noMinEps` (`booleantype`) flag controlling the bound on ϵ . If `SUNFALSE` is passed the value of ϵ is constrained and if `SUNTRUE` is passed then ϵ is not constrained.

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 ϵ (see `KINSetFuncNormTol` below).

KINSetMaxNewtonStep

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

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

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

`mxnewstep` (`realtype`) maximum scaled step length (≥ 0.0). Pass 0.0 to indicate the default.

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

`KIN_SUCCESS` The optional value has been successfully set.

`KIN_MEM_NULL` The `kin_mem` pointer is NULL.

`KIN_ILL_INPUT` The input value was negative.

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

KINSetMaxBetaFails

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

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

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

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

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

`KIN_SUCCESS` The optional value has been successfully set.

`KIN_MEM_NULL` The `kin_mem` pointer is NULL.

`KIN_ILL_INPUT` `mxnbcf` was negative.

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

KINSetRelErrFunc

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

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

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

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

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

`KIN_SUCCESS` The optional value has been successfully set.

`KIN_MEM_NULL` The `kin_mem` pointer is NULL.

`KIN_ILL_INPUT` The relative error was negative.

Notes The default value for `relfunc` is $U = \text{unit roundoff}$.

KINSetFuncNormTol

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

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

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

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

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

`KIN_SUCCESS` The optional value has been successfully set.

`KIN_MEM_NULL` The `kin_mem` pointer is NULL.

KIN_ILL_INPUT The tolerance was negative.

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

KINSetScaledStepTol

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

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

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`scsteptol` (`realtype`) tolerance for stopping based on scaled step length (≥ 0.0). Pass `0.0` to indicate the default.

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

KIN_SUCCESS The optional value has been successfully set.

KIN_MEM_NULL The `kin_mem` pointer is `NULL`.

KIN_ILL_INPUT The tolerance was non-positive.

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

KINSetConstraints

Call `flag = KINSetConstraints(kin_mem, constraints);`

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

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`constraints` (`N_Vector`) vector of constraint flags. If `constraints[i]` is
`0.0` then no constraint is imposed on u_i .
`1.0` then u_i will be constrained to be $u_i \geq 0.0$.
`-1.0` then u_i will be constrained to be $u_i \leq 0.0$.
`2.0` then u_i will be constrained to be $u_i > 0.0$.
`-2.0` then u_i will be constrained to be $u_i < 0.0$.

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

KIN_SUCCESS The optional value has been successfully set.

KIN_MEM_NULL The `kin_mem` pointer is `NULL`.

KIN_ILL_INPUT The constraint vector contains illegal values.

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

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

KINSetSysFunc

Call `flag = KINSetSysFunc(kin_mem, func);`

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

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`func` (`KINSysFn`) user-supplied function that evaluates $F(u)$ (or $G(u)$ for fixed-point iteration).

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

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

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

KINSetReturnNewest

Call `flag = KINSetReturnNewest(kin_mem, ret_newest);`

Description The function `KINSetReturnNewest` specifies if the fixed point iteration should return the newest iteration or the iteration consistent with the last function evaluation.

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

`SUNTRUE` – return the newest iteration.

`SUNFALSE` – return the iteration consistent with the last function evaluation.

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 of `ret_newest` is `SUNFALSE`.

KINSetDamping

Call `flag = KINSetDamping(kin_mem, beta);`

Description The function `KINSetDamping` specifies the value of the damping parameter in the fixed point or Picard iteration.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`beta` (`realtype`) the damping parameter value $0 < beta \leq 1.0$.

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

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

Notes This function sets the damping parameter value, which needs to be greater than zero and less than one if damping is to be used. A value ≥ 1 disables damping.

The default value of `beta` is 1.0, indicating no damping.

To set the damping parameter used in Anderson acceleration see `KINSetDampingAA`.

With the fixed point iteration the difference between successive iterations is used to determine convergence. As such, when damping is enabled, the tolerance used to stop the fixed point iteration is scaled by `beta` to account for the effects of damping. If `beta` is extremely small (close to zero), this can lead to an excessively tight tolerance.

KINSetMAA

Call `flag = KINSetMAA(kin_mem, maa);`

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

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`maa` (`long int`) subspace size for various methods. A value of 0 means no acceleration, while a positive value means acceleration will be done.

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

`KIN_SUCCESS` The optional value has been successfully set.

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

`KIN_ILL_INPUT` The argument `maa` was negative.

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

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

This function **MUST** be called before calling `KINInit`.

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

KINSetDampingAA

Call `flag = KINSetDampingAA(kin_mem, beta);`

Description The function `KINSetDampingAA` specifies the value of the Anderson acceleration damping parameter.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`beta` (`realtype`) the damping parameter value $0 < beta \leq 1.0$.

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

`KIN_SUCCESS` The optional value has been successfully set.

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

`KIN_ILL_INPUT` The argument `beta` was zero or negative.

Notes This function sets the damping parameter value, which needs to be greater than zero and less than one if damping is to be used. A value ≥ 1 disables damping.

The default value of `beta` is 1.0, indicating no damping.

When delaying the start of Anderson acceleration with `KINSetDelayAA`, use `KINSetDamping` to set the damping parameter in the fixed point or Picard iterations before Anderson acceleration begins. When using Anderson acceleration without delay, the value provided to `KINSetDampingAA` is applied to all iterations and any value provided to `KINSetDamping` is ignored.

KINSetDelayAA

Call `flag = KINSetDelayAA(kin_mem, delay);`

Description The function `KINSetDelayAA` specifies the number of iterations to delay the start of Anderson acceleration.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`delay` (`long int`) the number of iterations to delay Anderson acceleration.

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 `delay` was less than zero.

Notes The default value of `delay` is 0, indicating no delay.

4.5.4.2 Linear solver interface optional input functions

For matrix-based linear solver modules, the KINLS solver interface needs a function to compute an approximation to the Jacobian matrix $J(u)$. This function must be of type `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 KINLS solver. To specify a user-supplied Jacobian function `jac`, KINLS provides the function `KINSetJacFn`. The KINLS interface passes the pointer `user_data` to the Jacobian function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The pointer `user_data` may be specified through `KINSetUserData`.

`KINSetJacFn`

Call `flag = KINSetJacFn(kin_mem, jac);`

Description The function `KINSetJacFn` specifies the Jacobian approximation function to be used.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`jac` (`KINLsJacFn`) user-defined Jacobian approximation function.

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

KINLS_SUCCESS The optional 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 By default, KINLS uses an internal difference quotient function for dense and band matrices. If NULL is passed to `jac`, this default function is used. An error will occur if no `jac` is supplied when using a sparse or user-supplied matrix.

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

The function type `KINLsJacFn` is described in §4.6.4.

The previous routine `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 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 KINLS solver interface.

A user-defined Jacobian-vector function must be of type `KINLsJacTimesVecFn` and can be specified through a call to `KINLsSetJacTimesVecFn` (see §4.6.5 for specification details). The pointer `user_data` received through `KINSetUserData` (or a pointer to NULL if `user_data` was not specified) is passed to the Jacobian-times-vector function `jt看imes` 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.

`KINSetJacTimesVecFn`

Call `flag = KINSetJacTimesVecFn(kin_mem, jt看imes);`

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 value has 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 the internal difference quotient the user may optionally supply an alternative system function for use in the Jacobian-vector product approximation by calling `KINSetJacTimesVecSysFn`. The alternative system function should compute a suitable (and differentiable) approximation of the system function provided to `KINInit`. For example, as done in [20] when solving the nonlinear systems that arise in the implicit integration of ordinary differential equations, the alternative function may use lagged values when evaluating a nonlinearity to avoid differencing a potentially non-differentiable factor.

KINSetJacTimesVecSysFn

Call `flag = KINSetJacTimesVecSysFn(kin_mem, jtimesSysFn);`

Description The function `KINSetJacTimesVecSysFn` specifies an alternative system function for use in the internal Jacobian-vector product difference quotient approximation.

Arguments `kin_mem` (void *) pointer to the KINSOL memory block.
`jtimesSysFn` (KINSysFn) is the C function which computes the alternative system function to use in Jacobian-vector product difference quotient approximations. This function has the form `func(u, fval, user_data)`. (For full details see §4.6.1.)

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

KINLS_SUCCESS The optional value has been successfully set.
KINLS_MEM_NULL The `kin_mem` pointer is NULL.
KINLS_LMEM_NULL The KINLS linear solver has not been initialized.
KINLS_ILL_INPUT The internal difference quotient approximation is disabled.

Notes The default is to use the system function provided to `KINInit` in the internal difference quotient. If the input system function is NULL, the default is used.

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

F2003 Name `FKINSetJacTimesVecSysFn`

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.
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.

SUNDIALSGetVersion

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

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

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

Table 4.3: Optional outputs from KINSOL and KINLS

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

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

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

`SUNDIALSGetVersionNumber`

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

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

Arguments

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

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

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

4.5.5.2 Main solver optional output functions

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

KINGGetWorkspace

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

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

Arguments `kin_mem` (void *) pointer to the KINSOL memory block.
`lenrw` (long int) the number of `realtyp`e 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 $17+5N$ `realtyp`e 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).

KINGGetNumFuncEvals

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

Description The function `KINGGetNumFuncEvals` 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.

KINGGetNumNonlinSolvIters

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

Description The function `KINGGetNumNonlinSolvIters` 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.

KINGGetNumBetaCondFails

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

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

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

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

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

KINGGetNumBacktrackOps

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

Description The function `KINGGetNumBacktrackOps` 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`.

KINGGetFuncNorm

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

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

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

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

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

KINGGetStepLength

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

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

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.
`steplength` (`realtype`) scaled norm of the Newton step.

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

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

4.5.5.3 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.

KINGGetLinWorkSpace

Call `flag = KINGGetLinWorkSpace(kin_mem, &lenrwlS, &leniwlS);`

Description The function `KINGGetLinWorkSpace` 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.

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.

KINGGetNumLinIters

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

Description The function `KINGGetNumLinIters` 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.

KINGGetNumLinConvFails

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

Description The function `KINGGetNumLinConvFails` 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.

`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 `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.

KINGGetNumPrecEvals

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

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

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

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

`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 `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.

KINGGetNumPrecSolves

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

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

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

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

KINGGetNumJtimesEvals

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

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

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

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

KINGGetLastLinFlag

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

Description The function `KINGGetLastLinFlag` 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 `KIN_LSETUP_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`, `SUNLS_ASET_FAIL_UNREC`, or `SUNLS_PACKAGE_FAIL_UNREC`.

If the KINLS solve function failed (i.e., `KINsol` returned `KIN_LSOLVE_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_PSOLVE_FAIL_UNREC, indicating that the preconditioner solve function, `psolve`, failed with an unrecoverable error;
 SUNLS_GS_FAIL, indicating a failure in the Gram-Schmidt procedure (generated only in SPGMR or SPFGMR);
 SUNLS_QRSOL_FAIL, indicating that the matrix R was found to be singular during the QR solve phase (SPGMR and SPFGMR only); or
 SUNLS_PACKAGE_FAIL_UNREC, indicating an unrecoverable failure in an external iterative linear solver package.

The previous routines `KINDlsGetLastFlag` 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 `KINDlsGetReturnFlagName` 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 provides Jacobian-related information for the linear solver, and (optionally) one or two functions that define the preconditioner for use in any of the Krylov iterative algorithms.

4.6.1 Problem-defining function

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

KINSysFn

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

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

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

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

Notes Allocation of memory for `fval` is handled within KINSOL.

4.6.2 Error message handler function

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

`KINErrorHandlerFn`

Definition

```
typedef void (*KINErrorHandlerFn)(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 `KINSetErrorHandlerFn`.

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

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

4.6.3 Informational message handler function

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

`KINInfoHandlerFn`

Definition

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

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

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

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

4.6.4 Jacobian 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

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 8 for details).

With direct linear solvers (i.e., linear solvers with type `SUNLINEARSOLVER_DIRECT`), the Jacobian matrix $J(u)$ is zeroed out prior to calling the user-supplied Jacobian function 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`.

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 `SM_ELEMENT_D` and `SM_COLUMN_D` allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the `SUNMATRIX_DENSE` type. `SM_ELEMENT_D(J, i, j)` references the (i, j) -th element of the dense matrix `J` (with $i, j = 0 \dots N - 1$). This macro is meant for small problems for which efficiency of access is not a major concern. Thus, in terms of the indices m and n ranging from 1 to N , the Jacobian element $J_{m,n}$ can be set using the statement `SM_ELEMENT_D(J, m-1, n-1) = J_{m,n}`. Alternatively, `SM_COLUMN_D(J, j)` returns a pointer to the first element of the j -th column of `J` (with $j = 0 \dots N - 1$), and the elements of the j -th column can then be accessed using ordinary array indexing. Consequently, $J_{m,n}$ can be loaded using the statements `col_n = SM_COLUMN_D(J, n-1); col_n[m-1] = J_{m,n}`. For large problems, it is more efficient to use `SM_COLUMN_D` than to use `SM_ELEMENT_D`. Note that both of these macros number rows and columns starting from 0. The `SUNMATRIX_DENSE` type and accessor macros are documented in §8.3.

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 `SM_ELEMENT_B`, `SM_COLUMN_B`, and `SM_COLUMN_ELEMENT_B` allow the user to read and write banded matrix elements without making specific references to the underlying representation of the `SUNMATRIX_BAND` type. `SM_ELEMENT_B(J, i, j)` references the (i, j) -th element of the banded matrix `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 `mupper` and `mlower`, the Jacobian element $J_{m,n}$ can be loaded using the statement `SM_ELEMENT_B(J, m-1, n-1) = J_{m,n}`. The elements within the band are those with $-\text{mupper} \leq m-n \leq \text{mlower}$. Alternatively, `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 `realtype *col_j`, then the i -th element of the j -th column is given by `SM_COLUMN_ELEMENT_B(col_j, i, j)`, counting from 0. Thus, for (m, n) within the band, $J_{m,n}$ can be loaded by setting `col_n = SM_COLUMN_B(J, n-1)`; and `SM_COLUMN_ELEMENT_B(col_n, m-1, n-1) = J_{m,n}`. The elements of the j -th column can also be accessed via ordinary array indexing, but this approach requires knowledge of the underlying storage for a band matrix of type `SUNMATRIX_BAND`. The array `col_n` can be indexed from $-\text{mupper}$ to `mlower`. For large problems, it is more efficient to use `SM_COLUMN_B` and `SM_COLUMN_ELEMENT_B` than to use the `SM_ELEMENT_B` macro. As in the dense case, these macros all number rows and columns starting from 0. The `SUNMATRIX_BAND` type and accessor macros are documented in §8.4.

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 `SUNMATRIX_SPARSE` object may be accessed using the macro `SM_NNZ_S` or the routine `SUNSparseMatrix_NNZ`. The `SUNMATRIX_SPARSE` type and accessor macros are documented in §8.5.

The previous function type `KINDlsJacFn` is identical to `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, booleantype *new_u,
 void *user_data);`

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

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

- `user_data` is a pointer to user data, the same as the `user_data` parameter passed to `KINSetUserData`.
- Return value** The value returned by the Jacobian-times-vector function should be 0 if successful. If a recoverable failure occurred, the return value should be positive. In this case, KINSOL will attempt to correct by calling the preconditioner setup function. If this information is current, KINSOL halts. If the Jacobian-times-vector function encounters an unrecoverable error, it should return a negative value, prompting KINSOL to halt.
- Notes** If a user-defined routine is not given, then an internal `jtimes` function, using a difference quotient approximation, is used.
- 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:

`KINLsPrecSolveFn`

- Definition**

```
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 `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 §7.4.

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 `KINBBDDLocalFn`) which approximates the nonlinear system 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.

KINBBDDLocalFn

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

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

Arguments `Nlocal` is the local vector length.

`u` is the current value of the iterate.

`gval` is the output vector.

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

Return value A `KINBBDDLocalFn` function should return 0 if successful or a non-zero value if an error occurred.

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 int (*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 should return 0 if successful or a non-zero value if an error occurred.

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

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

to allocate memory for and initialize the internal preconditioner data. The last two arguments of `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 `KINSetPreconditioner` optional input functions.

10. Solve problem

11. Get optional output

Additional optional outputs associated with `KINBBDPRE` are available by way of two routines described below, `KINBBDPrecGetWorkSpace` and `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 KINBBDPRE (step 8), is described in more detail below.

KINBBDPrecInit

Call	<code>flag = KINBBDPrecInit(kin_mem, Nlocal, mudq, mldq, mukeep, mlkeep, dq_rel_u, Gloc, Gcomm);</code>
Description	The function <code>KINBBDPrecInit</code> initializes and allocates memory for the KINBBDPRE preconditioner.
Arguments	<p><code>kin_mem</code> (void *) pointer to the KINSOL memory block.</p> <p><code>Nlocal</code> (sunindextype) local vector length.</p> <p><code>mudq</code> (sunindextype) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.</p> <p><code>mldq</code> (sunindextype) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.</p> <p><code>mukeep</code> (sunindextype) upper half-bandwidth of the retained banded approximate Jacobian block.</p> <p><code>mlkeep</code> (sunindextype) lower half-bandwidth of the retained banded approximate Jacobian block.</p> <p><code>dq_rel_u</code> (realtype) the relative increment in components of <code>u</code> used in the difference quotient approximations. The default is <code>dq_rel_u = $\sqrt{\text{unit roundoff}}$</code>, which can be specified by passing <code>dq_rel_u = 0.0</code>.</p> <p><code>Gloc</code> (KINBBDDLocalFn) the C function which computes the approximation $G(u) \approx F(u)$.</p> <p><code>Gcomm</code> (KINBBDDCommFn) the optional C function which performs all interprocess communication required for the computation of $G(u)$.</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>KINLS_SUCCESS</code> The call to <code>KINBBDPrecInit</code> was successful.</p> <p><code>KINLS_MEM_NULL</code> The <code>kin_mem</code> pointer was <code>NULL</code>.</p> <p><code>KINLS_MEM_FAIL</code> A memory allocation request has failed.</p> <p><code>KINLS_LMEM_NULL</code> The KINLS linear solver interface has not been initialized.</p> <p><code>KINLS_ILL_INPUT</code> The supplied vector implementation was not compatible with the block band preconditioner.</p>
Notes	<p>If one of the half-bandwidths <code>mudq</code> or <code>mldq</code> to be used in the difference-quotient calculation of the approximate Jacobian is negative or exceeds the value <code>Nlocal-1</code>, it is replaced with 0 or <code>Nlocal-1</code> accordingly.</p> <p>The half-bandwidths <code>mudq</code> and <code>mldq</code> need not be the true half-bandwidths of the Jacobian of the local block of G, when smaller values may provide greater efficiency.</p> <p>Also, the half-bandwidths <code>mukeep</code> and <code>mlkeep</code> of the retained banded approximate Jacobian block may be even smaller, to reduce storage and computation costs further.</p> <p>For all four half-bandwidths, the values need not be the same for every process.</p> <p>The following two optional output functions are available for use with the KINBBDPRE module:</p>

KINBBDPrecGetWorkSpace

Call	<code>flag = KINBBDPrecGetWorkSpace(kin_mem, &lenrwBBDP, &leniwBBDP);</code>
Description	The function <code>KINBBDPrecGetWorkSpace</code> returns the local KINBBDPRE real and integer workspace sizes.

Arguments `kin_mem` (void *) pointer to the KINSOL memory block.
`lenrwBBDP` (long int) local number of `realt` values in the KINBBDPRE workspace.
`leniwBBDP` (long int) local number of integer values in the KINBBDPRE workspace.

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

Notes The workspace requirements reported by this routine correspond only to memory allocated within the KINBBDPRE module (the banded matrix approximation, banded SUNLINSOL object, temporary vectors). These values are local to each process.
The workspaces referred to here exist in addition to those given by the corresponding `KINGetLinWorkSpace` function.

KINBBDPrecGetNumGfnEvals

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

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

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

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

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

Chapter 5

Using KINSOL for Fortran Applications

A Fortran 2003 module (`fkinsol_mod`) as well as a Fortran 77 style interface (`FKINSOL`) are provided to support the use of KINSOL, 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.

5.1 KINSOL Fortran 2003 Interface Module

The `fkinsol_mod` Fortran module defines interfaces to most KINSOL C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. All interfaced functions are named after the corresponding C function, but with a leading 'F'. For example, the KINSOL function `KINcreate` is interfaced as `FKINcreate`. Thus, the steps to use KINSOL and the function calls in Fortran 2003 are identical (ignoring language differences) to those in C. The C functions with Fortran 2003 interfaces indicate this in their description in Chapter 4. The Fortran 2003 KINSOL interface module can be accessed by the `use` statement, i.e. `use fkinsol_mod`, and linking to the library `libsundials_fkinsol_mod.lib` in addition to `libsundials_kinsol.lib`.

The Fortran 2003 interface modules were generated with SWIG Fortran, a fork of SWIG [29]. Users who are interested in the SWIG code used in the generation process should contact the SUNDIALS development team.

5.1.1 SUNDIALS Fortran 2003 Interface Modules

All of the generic SUNDIALS modules provide Fortran 2003 interface modules. Many of the generic module implementations provide Fortran 2003 interfaces (a complete list of modules with Fortran 2003 interfaces is given in Table 5.1). A module can be accessed with the `use` statement, e.g. `use fnvector_openmp_mod`, and linking to the Fortran 2003 library in addition to the C library, e.g. `libsundials_fnvecopenmp_mod.lib` and `libsundials_nvecopenmp.lib`.

The Fortran 2003 interfaces leverage the `iso_c_binding` module and the `bind(C)` attribute to closely follow the SUNDIALS C API (ignoring language differences). The generic SUNDIALS structures, e.g. `N_Vector`, are interfaced as Fortran derived types, and function signatures are matched but with an `F` prepending the name, e.g. `FN_VConst` instead of `N_VConst`. Constants are named exactly as they are in the C API. Accordingly, using SUNDIALS via the Fortran 2003 interfaces looks just like using it in C. Some caveats stemming from the language differences are discussed in the section 5.1.3. A discussion on the topic of equivalent data types in C and Fortran 2003 is presented in section 5.1.2.

Further information on the Fortran 2003 interfaces specific to modules is given in the `NVECTOR`, `SUNMATRIX`, `SUNLINSOL`, and `SUNNONLINSOL` alongside the C documentation (chapters 7, 8, 9, and

?? respectively). For details on where the Fortran 2003 module (.mod) files and libraries are installed see Appendix A.

Table 5.1: Summary of Fortran 2003 interfaces for shared SUNDIALS modules.

Module	Fortran 2003 Module Name
NVECTOR	fsundials_nvector_mod
NVECTOR_SERIAL	fnvector_serial_mod
NVECTOR_PARALLEL	fnvector_parallel_mod
NVECTOR_OPENMP	fnvector_openmp_mod
NVECTOR_PTHREADS	fnvector_pthreads_mod
NVECTOR_PARHYP	Not interfaced
NVECTOR_PETSC	Not interfaced
NVECTOR_CUDA	Not interfaced
NVECTOR_RAJA	Not interfaced
NVECTOR_MANYVECTOR	fnvector_manyvector_mod
NVECTOR_MPIMANYVECTOR	fnvector_mpimanyvector_mod
NVECTOR_MPIPLUSX	fnvector_mpiplusx_mod
SUNMatrix	fsundials_matrix_mod
SUNMATRIX_BAND	fsunmatrix_band_mod
SUNMATRIX_DENSE	fsunmatrix_dense_mod
SUNMATRIX_SPARSE	fsunmatrix_sparse_mod
SUNLinearSolver	fsundials_linearsolver_mod
SUNLINSOL_BAND	fsunlinsol_band_mod
SUNLINSOL_DENSE	fsunlinsol_dense_mod
SUNLINSOL_LAPACKBAND	Not interfaced
SUNLINSOL_LAPACKDENSE	Not interfaced
SUNLINSOL_KLU	fsunlinsol_klu_mod
SUNLINSOL_SUPERLUMT	Not interfaced
SUNLINSOL_SUPERLUDIST	Not interfaced
SUNLINSOL_SPGMR	fsunlinsol_spgmr_mod
SUNLINSOL_SPGMR	fsunlinsol_spgmr_mod
SUNLINSOL_SPBCGS	fsunlinsol_spbcgs_mod
SUNLINSOL_SPTFQMR	fsunlinsol_sptfqmr_mod
SUNLINSOL_PCG	fsunlinsol_pcg_mod
SUNNonlinearSolver	fsundials_nonlinearsolver_mod
SUNNONLINSOL_NEWTON	fsunnonlinsol_newton_mod
SUNNONLINSOL_FIXEDPOINT	fsunnonlinsol_fixedpoint_mod

5.1.2 Data Types

Generally, the Fortran 2003 type that is equivalent to the C type is what one would expect. Primitive types map to the `iso_c_binding` type equivalent. SUNDIALS generic types map to a Fortran derived type. However, the handling of pointer types is not always clear as they can depend on the parameter direction. Table 5.2 presents a summary of the type equivalencies with the parameter direction in mind.



Currently, the Fortran 2003 interfaces are only compatible with SUNDIALS builds where the `realtyp` is double precision and the `sunindextyp` size is 64-bits.

Table 5.2: C/Fortran 2003 Equivalent Types

C type	Parameter Direction	Fortran 2003 type
double	in, inout, out, return	real(c_double)
int	in, inout, out, return	integer(c_int)
long	in, inout, out, return	integer(c_long)
booleantype	in, inout, out, return	integer(c_int)
realttype	in, inout, out, return	real(c_double)
sunindextype	in, inout, out, return	integer(c_long)
double*	in, inout, out	real(c_double), dimension(*)
double*	return	real(c_double), pointer, dimension(:)
int*	in, inout, out	integer(c_int), dimension(*)
int*	return	integer(c_int), pointer, dimension(:)
long*	in, inout, out	integer(c_long), dimension(*)
long*	return	integer(c_long), pointer, dimension(:)
realttype*	in, inout, out	real(c_double), dimension(*)
realttype*	return	real(c_double), pointer, dimension(:)
sunindextype*	in, inout, out	integer(c_long), dimension(*)
sunindextype*	return	integer(c_long), pointer, dimension(:)
realttype[]	in, inout, out	real(c_double), dimension(*)
sunindextype[]	in, inout, out	integer(c_long), dimension(*)
N_Vector	in, inout, out	type(N_Vector)
N_Vector	return	type(N_Vector), pointer
SUNMatrix	in, inout, out	type(SUNMatrix)
SUNMatrix	return	type(SUNMatrix), pointer
SUNLinearSolver	in, inout, out	type(SUNLinearSolver)
SUNLinearSolver	return	type(SUNLinearSolver), pointer
SUNNonlinearSolver	in, inout, out	type(SUNNonlinearSolver)
SUNNonlinearSolver	return	type(SUNNonlinearSolver), pointer
FILE*	in, inout, out, return	type(c_ptr)
void*	in, inout, out, return	type(c_ptr)
T**	in, inout, out, return	type(c_ptr)
T***	in, inout, out, return	type(c_ptr)
T****	in, inout, out, return	type(c_ptr)

5.1.3 Notable Fortran/C usage differences

While the Fortran 2003 interface to SUNDIALS closely follows the C API, some differences are inevitable due to the differences between Fortran and C. In this section, we note the most critical differences. Additionally, section 5.1.2 discusses equivalencies of data types in the two languages.

5.1.3.1 Creating generic SUNDIALS objects

In the C API a generic SUNDIALS object, such as an `N_Vector`, is actually a pointer to an underlying C struct. However, in the Fortran 2003 interface, the derived type is bound to the C struct, not the pointer to the struct. E.g., `type(N_Vector)` is bound to the C struct `_generic_N_Vector` not the `N_Vector` type. The consequence of this is that creating and declaring SUNDIALS objects in Fortran is nuanced. This is illustrated in the code snippets below:

C code:

```
N_Vector x;
x = N_VNew_Serial(N);
```

Fortran code:

```
type(N_Vector), pointer :: x
x => FN_VNew_Serial(N)
```

Note that in the Fortran declaration, the vector is a `type(N_Vector), pointer`, and that the pointer assignment operator is then used.

5.1.3.2 Arrays and pointers

Unlike in the C API, in the Fortran 2003 interface, arrays and pointers are treated differently when they are return values versus arguments to a function. Additionally, pointers which are meant to be out parameters, not arrays, in the C API must still be declared as a rank-1 array in Fortran. The reason for this is partially due to the Fortran 2003 standard for C bindings, and partially due to the tool used to generate the interfaces. Regardless, the code snippets below illustrate the differences.

C code:

```
N_Vector x
realtype* xdata;
long int leniw, lenrw;

x = N_VNew_Serial(N);

/* capturing a returned array/pointer */
xdata = N_VGetArrayPointer(x)

/* passing array/pointer to a function */
N_VSetArrayPointer(xdata, x)

/* pointers that are out-parameters */
N_VSpace(x, &leniw, &lenrw);
```

Fortran code:

```
type(N_Vector), pointer :: x
real(c_double), pointer :: xdataptr(:)
real(c_double)          :: xdata(N)
integer(c_long)         :: leniw(1), lenrw(1)

x => FN_VNew_Serial(x)

! capturing a returned array/pointer
xdataptr => FN_VGetArrayPointer(x)

! passing array/pointer to a function
call FN_VSetArrayPointer(xdata, x)

! pointers that are out-parameters
call FN_VSpace(x, leniw, lenrw)
```

5.1.3.3 Passing procedure pointers and user data

Since functions/subroutines passed to SUNDIALS will be called from within C code, the Fortran procedure must have the attribute `bind(C)`. Additionally, when providing them as arguments to a Fortran 2003 interface routine, it is required to convert a procedure's Fortran address to C with the Fortran intrinsic `c_funloc`.

Typically when passing user data to a SUNDIALS function, a user may simply cast some custom data structure as a `void*`. When using the Fortran 2003 interfaces, the same thing can be achieved.

Note, the custom data structure *does not* have to be `bind(C)` since it is never accessed on the C side.

C code:

```
MyUserData* udata;
void *ckode_mem;

ierr = CkdeSetUserData(ckode_mem, udata);
```

Fortran code:

```
type(MyUserData) :: udata
type(c_ptr)      :: ckode_mem

ierr = FCkdeSetUserData(ckode_mem, c_loc(udata))
```

On the other hand, Fortran users may instead choose to store problem-specific data, e.g. problem parameters, within modules, and thus do not need the SUNDIALS-provided `user_data` pointers to pass such data back to user-supplied functions. These users should supply the `c_null_ptr` input for `user_data` arguments to the relevant SUNDIALS functions.

5.1.3.4 Passing NULL to optional parameters

In the SUNDIALS C API some functions have optional parameters that a caller can pass NULL to. If the optional parameter is of a type that is equivalent to a Fortran `type(c_ptr)` (see section 5.1.2), then a Fortran user can pass the intrinsic `c_null_ptr`. However, if the optional parameter is of a type that is not equivalent to `type(c_ptr)`, then a caller must provide a Fortran pointer that is dissociated. This is demonstrated in the code example below.

C code:

```
SUNLinearSolver LS;
N_Vector x, b;

! SUNLinSolSolve expects a SUNMatrix or NULL
! as the second parameter.
ierr = SUNLinSolSolve(LS, NULL, x, b);
```

Fortran code:

```
type(SUNLinearSolver), pointer :: LS
type(SUNMatrix), pointer :: A
type(N_Vector), pointer :: x, b

A => null()

! SUNLinSolSolve expects a type(SUNMatrix), pointer
! as the second parameter. Therefore, we cannot
! pass a c_null_ptr, rather we pass a disassociated A.
ierr = FSUNLinSolSolve(LS, A, x, b)
```

5.1.3.5 Working with N_Vector arrays

Arrays of `N_Vector` objects are interfaced to Fortran 2003 as opaque `type(c_ptr)`. As such, it is not possible to directly index an array of `N_Vector` objects returned by the `N_Vector` “VectorArray” operations, or packages with sensitivity capabilities. Instead, SUNDIALS provides a utility function `FN_VGetVecAtIndexVectorArray` that can be called for accessing a vector in a vector array. The

example below demonstrates this:

C code:

```
N_Vector x;
N_Vector* vecs;

vecs = N_VCloneVectorArray(count, x);
for (int i=0; i < count; ++i)
    N_VConst(vecs[i]);
```

Fortran code:

```
type(N_Vector), pointer :: x, xi
type(c_ptr)           :: vecs

vecs = FN_VCloneVectorArray(count, x)
do index, count
    xi => FN_VGetVecAtIndexVectorArray(vecs, index)
    call FN_VConst(xi)
enddo
```

SUNDIALS also provides the functions `FN_VSetVecAtIndexVectorArray` and `FN_VNewVectorArray` for working with `N_Vector` arrays. These functions are particularly useful for users of the Fortran interface to the `NVECTOR_MANYVECTOR` or `NVECTOR_MPIMANYVECTOR` when creating the subvector array. Both of these functions along with `FN_VGetVecAtIndexVectorArray` are further described in Chapter 7.1.6.

5.1.3.6 Providing file pointers

Expert SUNDIALS users may notice that there are a few advanced functions in the SUNDIALS C API that take a `FILE *` argument. Since there is no portable way to convert between a Fortran file descriptor and a C file pointer, SUNDIALS provides two utility functions for creating a `FILE *` and destroying it. These functions are defined in the module `fsundials_futils_mod`.

FSUNDIALSFileOpen

Call `fp = FSUNDIALSFileOpen(filename, mode)`

Description The function allocates a `FILE *` by calling the C function `fopen`.

Arguments `filename` (`character(kind=C_CHAR, len=*)`) - the path to the file to open
`mode` (`character(kind=C_CHAR, len=*)`) - the mode string given to `fopen`. It should begin with one of the following characters:

“r” - open text file for reading

“r+” - open text file for reading and writing

“w” - truncate text file to zero length or create it for writing

“w+” - open text file for reading or writing, create it if it does not exist

“a” - open for appending, see documentation of “fopen” for your system/compiler

“a+” - open for reading and appending, see documentation for “fopen” for your system/compiler

Return value This returns a `type(C_PTR)` which is a `FILE*` in C. If it is `NULL`, then there was an error opening the file.

FSUNDIALSFileClose

Call call FSUNDIALSFileClose(fp)

Description The function deallocates a FILE* by calling the C function `fclose`.

Arguments fp (type(C_PTR)) - the file pointer (type FILE* in C)

Return value None

5.1.4 Important notes on portability

The SUNDIALS Fortran 2003 interface *should* be compatible with any compiler supporting the Fortran 2003 ISO standard. However, it has only been tested and confirmed to be working with GNU Fortran 4.9+ and Intel Fortran 18.0.1+.

Upon compilation of SUNDIALS, Fortran module (.mod) files are generated for each Fortran 2003 interface. These files are highly compiler specific, and thus it is almost always necessary to compile a consuming application with the same compiler used to generate the modules.

5.1.5 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.

5.2 Important note on portability

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

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

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

5.3 Fortran Data Types

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

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

- `int32_t` – equivalent to an `INTEGER` or `INTEGER*4` in FORTRAN
- `int64_t` – equivalent to an `INTEGER*8` in FORTRAN

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

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

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

5.3.1 FKINSOL routines

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

- Interface to the NVECTOR modules
 - `FNVINITS` (defined by `NVECTOR_SERIAL`) interfaces to `N_VNewEmpty_Serial`.
 - `FNVINITP` (defined by `NVECTOR_PARALLEL`) interfaces to `N_VNewEmpty_Parallel`.
 - `FNVINITOMP` (defined by `NVECTOR_OPENMP`) interfaces to `N_VNewEmpty_OpenMP`.
 - `FNVINITPTS` (defined by `NVECTOR_PTHREADS`) interfaces to `N_VNewEmpty_Pthreads`.
- Interface to the SUNMATRIX modules
 - `FSUNBANDMATINIT` (defined by `SUNMATRIX_BAND`) interfaces to `SUNBandMatrix`.
 - `FSUNDENSEMATINIT` (defined by `SUNMATRIX_DENSE`) interfaces to `SUNDenseMatrix`.
 - `FSUNSPARSEMATINIT` (defined by `SUNMATRIX_SPARSE`) interfaces to `SUNSparseMatrix`.
- Interface to the SUNLINSOL modules
 - `FSUNBANDLINSOLINIT` (defined by `SUNLINSOL_BAND`) interfaces to `SUNLinSol_Band`.
 - `FSUNDENSELINSOLINIT` (defined by `SUNLINSOL_DENSE`) interfaces to `SUNLinSol_Dense`.
 - `FSUNKLUINIT` (defined by `SUNLINSOL_KLU`) interfaces to `SUNLinSol_KLU`.
 - `FSUNKLUREINIT` (defined by `SUNLINSOL_KLU`) interfaces to `SUNLinSol_KLUReinit`.
 - `FSUNLAPACKBANDINIT` (defined by `SUNLINSOL_LAPACKBAND`) interfaces to `SUNLinSol_LapackBand`.
 - `FSUNLAPACKDENSEINIT` (defined by `SUNLINSOL_LAPACKDENSE`) interfaces to `SUNLinSol_LapackDense`.
 - `FSUNPCGINIT` (defined by `SUNLINSOL_PCG`) interfaces to `SUNLinSol_PCG`.
 - `FSUNSPBCGSINIT` (defined by `SUNLINSOL_SPBCGS`) interfaces to `SUNLinSol_SPBCGS`.
 - `FSUNSPFGMRINIT` (defined by `SUNLINSOL_SPGMR`) interfaces to `SUNLinSol_SPGMR`.
 - `FSUNSPGMRINIT` (defined by `SUNLINSOL_SPGMR`) interfaces to `SUNLinSol_SPGMR`.
 - `FSUNSPTFQMRINIT` (defined by `SUNLINSOL_SPTFQMR`) interfaces to `SUNLinSol_SPTFQMR`.
 - `FSUNSUPERLUMTINIT` (defined by `SUNLINSOL_SUPERLUMT`) interfaces to `SUNLinSol_SuperLUMT`.
- Interface to the main KINSOL module
 - `FKINCREATE` interfaces to `KINCreate`.

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

FKINSOL routine (FORTRAN, user-supplied)	KINSOL function (C, interface)	KINSOL type of interface function
FKFUN	FKINfunc	KINSysFn
FKDJAC	FKINDenseJac	KINLsJacFn
FKBJAC	FKINBandJac	KINLsJacFn
FKINSPJAC	FKINSparsJac	KINLsJacFn
FKPSET	FKINPSet	KINLsPrecSetupFn
FKPSOL	FKINPSol	KINLsPrecSolveFn
FKJTIMES	FKINJtimes	KINLsJacTimesVecFn

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.

5.3.2 Usage of the FKINSOL interface module

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

1. Nonlinear system function specification

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

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

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

2. NVECTOR module initialization

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

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

in which the name and call sequence are as described in the appropriate section of Chapter 7.

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

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

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

4. SUNLINSOL module initialization

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

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

in which the call sequence is as described in the appropriate section of Chapter 9. Note that the dense, band, or sparse solvers are usable only in a serial or multi-threaded environment.

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

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

where again the call sequences are described in the appropriate sections of Chapter 9.

5. Problem specification

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

FKINCREATE

Call CALL FKINCREATE (IER)

Description This function creates the KINSOL memory structure.

Arguments None.

Return value IER is the return completion flag. Values are 0 for successful return and -1 otherwise. See printed message for details in case of failure.

Notes

6. Set optional inputs

Call FKINSETIIN, FKINSETRIN, and/or FKINSETVIN, to set desired optional inputs, if any. See §5.3.3 for details.

7. Solver Initialization

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

FKININIT

Call CALL FKININIT (IOUT, ROUT, IER)

Description This function specifies the optional output arrays, allocates internal memory, and initializes KINSOL.

Arguments IOUT is an integer array for integer optional outputs.
ROUT is a real array for real optional outputs.

Return value IER is the return completion flag. Values are 0 for successful return and -1 otherwise. See printed message for details in case of failure.

Notes The user integer data array IOUT must be declared as INTEGER*4 or INTEGER*8 according to the C type long int.
The optional outputs associated with the main KINSOL integrator are listed in Table 5.4.

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:

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:

```
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:

```
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:

```
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 $k = i - j + MU + 1$ ($k = 1 \dots ML + MU + 1$) and $j = 1 \dots N$. 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:

```
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 `sunmatrix_sparse.h` for more information). The arguments are `Y`, an array containing state variables; `FY`, an array containing residual values; `N`, the number of matrix rows/columns in the Jacobian; `NNZ`, allocated length of nonzero storage; `JDATA`, nonzero values in the Jacobian (of length `NNZ`); `JINDEXVALS`, row [or column] indices for each nonzero in Jacobian (of length `NNZ`); `JINDEXPTRS`, pointers to each Jacobian column [or row] in the two preceding arrays (of length `N+1`); `WK*`, work arrays containing temporary workspace of same size as `Y`; and `IER`, error return code (0 if successful, > 0 if a recoverable error occurred, or < 0 if an unrecoverable error occurred.)

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

```
CALL FKINSPARSESETJAC (IER)
```

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

KINLS with Jacobian-vector product

As an option when using the `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:

```

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

```

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

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

```
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 `FKINSPILSETJAC` 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:

```

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

```

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

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

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

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

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

To indicate that the `FKINPSET` and `FKINPSOL` routines are supplied, then the user must call

```
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 `FKPSOL` and `FKPSET` (see below).

The previous routine `FKINSPILSETPREC` 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 `FKINLSSETPREC`, the routine `FKPSET` must be provided, even if it is not needed, and then it should return `IER = 0`.

9. Problem solution

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

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

The arguments are as follows. `U` is an array containing the initial guess on input, and the solution on return. `GLOBALSTRAT` is an integer (type `INTEGER`) defining the global strategy choice (0 specifies Inexact Newton, 1 indicates Newton with line search, 2 indicates Picard iteration, and 3 indicates Fixed Point iteration). `USCALE` is an array of scaling factors for the `U` vector. `FSCALE` is an array of scaling factors for the `FVAL` vector. `IER` is an integer completion flag and will have one of the following values: 0 to indicate success, 1 to indicate that the initial guess satisfies $F(u) = 0$ within tolerances, 2 to indicate apparent stalling (small step), or a negative value to indicate an error or failure. These values correspond to the `KINSOL` returns (see §4.5.3 and §B.2). The values of the optional outputs are available in `IOPT` and `ROPT` (see Table 5.4).

10. Memory deallocation

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

```
CALL FKINFREE
```

Table 5.3: Keys for setting FKINSOL optional inputs

Integer optional inputs FKINSETIIN		
Key	Optional input	Default value
PRNT_LEVEL	Verbosity level of output	0
MAA	Number of prior residuals for Anderson Acceleration	0
MAX_NITERS	Maximum no. of nonlinear iterations	200
ETA_FORM	Form of η coefficient	1 (KIN_ETACHOICE1)
MAX_SETUPS	Maximum no. of iterations without prec. setup	10
MAX_SP_SETUPS	Maximum no. of iterations without residual check	5
NO_INIT_SETUP	No initial preconditioner setup	SUNFALSE
NO_MIN_EPS	Lower bound on ϵ	SUNFALSE
NO_RES_MON	No residual monitoring	SUNFALSE

Real optional inputs (FKINSETRIN)		
Key	Optional input	Default value
FNORM_TOL	Function-norm stopping tolerance	$\text{uround}^{1/3}$
SSTEP_TOL	Scaled-step stopping tolerance	$\text{uround}^{2/3}$
MAX_STEP	Max. scaled length of Newton step	$1000\ D_u u_0\ _2$
RERR_FUNC	Relative error for F.D. Jv	$\sqrt{\text{uround}}$
ETA_CONST	Constant value of η	0.1
ETA_PARAMS	Values of γ and α	0.9 and 2.0
RMON_CONST	Constant value of ω	0.9
RMON_PARAMS	Values of ω_{min} and ω_{max}	0.00001 and 0.9

5.3.3 FKINSOL optional input and output

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

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

where KEY is a quoted string indicating which optional input is set, IVAL is the integer input value to be used, RVAL is the real input value to be used, and VVAL is the input real array to be used. IER is an integer return flag which is set to 0 on success and a negative value if a failure occurred. For the legal values of KEY in calls to FKINSETIIN and FKINSETRIN, see Table 5.3. The one legal value of KEY for FKINSETVIN is CONSTR_VEC, for providing the array of inequality constraints to be imposed on the solution, if any. The integer IVAL should be declared in a manner consistent with C type `long int`.

The optional outputs from the KINSOL solver are accessed not through individual functions, but rather through a pair of arrays, IOUT (integer type) of dimension at least 15, and ROUT (real type) of dimension at least 2. These arrays are owned (and allocated) by the user and are passed as arguments to FKININIT. Table 5.4 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.

5.3.4 Usage of the FKINBBD interface to KINBBDPRE

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

Table 5.4: Description of the FKINSOL optional output arrays IOU and ROUT

Integer output array IOU		
Index	Optional output	KINSOL function
KINSOL main solver		
1	LENRW	KINGetWorkSpace
2	LENIW	KINGetWorkSpace
3	NNI	KINGetNumNonlinSolvIters
4	NFE	KINGetNumFuncEvals
5	NBCF	KINGetNumBetaCondFails
6	NBKTRK	KINGetNumBacktrackOps
KINLS linear solver interface		
7	LENRWLS	KINGetLinWorkSpace
8	LENIWLS	KINGetLinWorkSpace
9	LS_FLAG	KINGetLastLinFlag
10	NFELS	KINGetNumLinFuncEvals
11	NJE	KINGetNumJacEvals
12	NJTV	KINGetNumJtimesEvals
13	NPE	KINGetNumPrecEvals
14	NPS	KINGetNumPrecSolves
15	NLI	KINGetNumLinIters
16	NCFL	KINGetNumLinConvFails

Real output array ROUT		
Index	Optional output	KINSOL function
1	FNORM	KINGetFuncNorm
2	SSTEP	KINGetStepLength

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

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

In addition to the FORTRAN right-hand side function FKFUN, the user-supplied functions used by this package, are listed below, each with the corresponding interface function which calls it (and its type within KINBBDPRE or KINSOL):

FKINBBD routine (FORTRAN, user-supplied)	KINSOL function (C, interface)	KINSOL type of interface function
FKLOCFN	FKINGloc	KINBBDDLocalFn
FKCOMMF	FKINGcomm	KINBBDDCommFn
FKJTIMES	FKINJtimes	KINLsJacTimesVecFn

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 §5.3.1, 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 §5.3.2 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:

```
CALL FKINBBDINIT (NLOCAL, MUDQ, MLDQ, MU, ML, IER)
```

The arguments are as follows. NLOCAL is the local size of vectors for this process. MUDQ and MLDQ are the upper and lower half-bandwidths to be used in the computation of the local Jacobian blocks 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 `FLAG` $\neq 0$. (See step 8 in §5.3.2).

8. Problem solution

9. KINBBDPRE **Optional outputs**

Optional outputs specific to the SPGMR, SPFGMR, SPBCGS, or SPTFQMR solver are listed in Table 5.4. To obtain the optional outputs associated with the KINBBDPRE module, make the following call:

```
CALL FKINBBDOPT (LENRBBD, LENIBBD, NGEBBD)
```

The arguments should be consistent with C type `long int`. Their returned values are as follows: `LENRBBD` is the length of real preconditioner work space, in `realtype` words. `LENIBBD` is the length of integer preconditioner work space, in integer words. These sizes are local to the current process. `NGEBBD` is the cumulative number of $G(u)$ evaluations (calls to `FKLOCFN`) so far.

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:

```
SUBROUTINE FKLOCFN (NLOC, ULOC, GLOC, IER)
  DIMENSION ULOC(*), GLOC(*)
```

This routine is to evaluate the function $G(u)$ approximating F (possibly identical to F), in terms of the array `ULOC` (of length `NLOC`), which is the sub-vector of u local to this processor. The resulting (local) sub-vector is to be stored in the array `GLOC`. `IER` is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if `FKLOCFN` failed unrecoverably (in which case the solution process is halted).

```
SUBROUTINE FKCOMMFN (NLOC, ULOC, IER)
  DIMENSION ULOC(*)
```

This routine is to perform the inter-processor communication necessary for the `FKLOCFN` routine. Each call to `FKCOMMFN` is preceded by a call to the system function routine `FKFUN` with the same argument `ULOC`. `IER` is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if `FKCOMMFN` failed recoverably (in which case the solution process is halted).



The subroutine `FKCOMMFN` must be supplied even if it is not needed and must return `IER = 0`.

Optionally, the user can supply a routine `FKINJTIMES` for the evaluation of Jacobian-vector products, as described above in step 8 in §5.3.2. Note that this routine is required if using Picard iteration.

Chapter 6

KINSOL Features for GPU Accelerated Computing

This chapter is concerned with using GPU-acceleration and KINSOL for the solution of systems of nonlinear algebraic equations.

6.1 SUNDIALS GPU Programming Model

In this section, we introduce the SUNDIALS GPU programming model and highlight SUNDIALS GPU features. The model leverages the fact that all of the SUNDIALS packages interact with simulation data either through the shared vector, matrix, and solver APIs (see §7, §8, §9, and §??) or through user-supplied callback functions. Thus, under the model, the overall structure of the user's calling program, and the way users interact with the SUNDIALS packages is similar to using SUNDIALS in CPU-only environments.

Within the SUNDIALS GPU programming model, all control logic executes on the CPU, and all simulation data resides wherever the vector or matrix object dictates as long as SUNDIALS is in control of the program. That is, SUNDIALS will not migrate data (explicitly) from one memory space to another. Except in the most advanced use cases, it is safe to assume that data is kept resident in the GPU-device memory space. The consequence of this is that, when control is passed from the user's calling program to SUNDIALS, simulation data in vector or matrix objects must be up-to-date in the device memory space. Similarly, when control is passed from SUNDIALS to the user's calling program, the user should assume that any simulation data in vector and matrix objects are up-to-date in the device memory space. To put it succinctly, *it is the responsibility of the user's calling program to manage data coherency between the CPU and GPU-device memory spaces* unless unified virtual memory (UVM), also known as managed memory, is being utilized. Typically, the GPU-enabled SUNDIALS modules provide functions to copy data from the host to the device and vice-versa as well as support for unmanaged memory or UVM. In practical terms, the way SUNDIALS handles distinct host and device memory spaces means that *users need to ensure that the user-supplied functions, e.g. the right-hand side function, only operate on simulation data in the device memory space* otherwise extra memory transfers will be required and performance will be poor. The exception to this rule is if some form of hybrid data partitioning (achievable with the `NVECTOR_MANYVECTOR` §7.15) is utilized.

SUNDIALS provides many native shared features and modules that are GPU-enabled. Currently, these are primarily limited to the NVIDIA CUDA platform [5], although support for more GPU computing platforms such as AMD ROCm/HIP [1] and Intel oneAPI [2], is an area of active development. Table 6.1 summarizes the shared SUNDIALS modules that are GPU-enabled, what GPU programming environments they support, and what class of memory they support (unmanaged or UVM). Users may also supply their own GPU-enabled `N.Vector`, `SUNMatrix`, `SUNLinearSolver`, or `SUNNonlinearSolver` implementation, and the capabilities will be leveraged since SUNDIALS operates on data through these APIs.

In addition, SUNDIALS provides the `SUNMemoryHelper` API §10.1 to support applications which implement their own memory management or memory pooling.

Table 6.1: List of SUNDIALS GPU Enabled Modules. Note that support for ROCm/HIP and oneAPI are currently untested, and implicit UVM (i.e. `malloc` returning UVM) is not accounted for. A The † symbol indicates that the module inherits support from the NVECTOR module used.

Module	CUDA	ROCm/HIP	oneAPI	Unmanaged memory	UVM
NVECTOR_CUDA (§7.9)	✓			✓	✓
NVECTOR_RAJA (§7.11)	✓			✓	✓
NVECTOR_OPENMPDEV (§7.13)	✓	✓	✓	✓	
SUNMATRIX_CUSPARSE (§8.7)	✓			✓	✓
SUNLINSOL_CUSOLVERS_BATCHQR (§9.12)	✓			✓	✓
SUNLINSOL_SPGMR (§9.15)	†	†	†	†	†
SUNLINSOL_SPGMR (§9.16)	†	†	†	†	†
SUNLINSOL_SPTFQMR (§9.18)	†	†	†	†	†
SUNLINSOL_SPBCGS (§9.17)	†	†	†	†	†
SUNLINSOL_PCG (§9.19)	†	†	†	†	†
SUNNONLINSOL_NEWTON (§??)	†	†	†	†	†
SUNNONLINSOL_FIXEDPOINT (§??)	†	†	†	†	†

6.2 Steps for Using GPU Accelerated SUNDIALS

For any SUNDIALS package, the generalized steps a user needs to take to use GPU accelerated SUNDIALS are:

1. Utilize a GPU-enabled NVECTOR implementation. Initial data can be loaded on the host, but must be in the device memory space prior to handing control to SUNDIALS.
2. Utilize a GPU-enabled SUNLINSOL linear solver (if necessary).
3. Utilize a GPU-enabled SUNMATRIX implementation (if using a matrix-based linear solver).
4. Utilize a GPU-enabled SUNNONLINSOL nonlinear solver (if necessary).
5. Write user-supplied functions so that they use data only in the device memory space (again, unless an atypical data partitioning is used). A few examples of these functions are the right-hand side evaluation function, the Jacobian evaluation function, or the preconditioner evaluation function. In the context of CUDA and the right-hand side function, one way a user might ensure data is accessed on the device is, for example, calling a CUDA kernel, which does all of the computation, from a CPU function which simply extracts the underlying device data array from the NVECTOR object that is passed from SUNDIALS to the user-supplied function.

Users should refer to Table 6.1 for a list of GPU-enabled native SUNDIALS modules.

Chapter 7

Description of the NVECTOR module

The SUNDIALS solvers are written in a data-independent manner. They all operate on generic vectors (of type `N_Vector`) through a set of operations defined by the particular NVECTOR implementation. Users can provide their own specific implementation of the NVECTOR module, or use one of the implementations provided with SUNDIALS. The generic NVECTOR is described below and the implementations provided with SUNDIALS are described in the following sections.

7.1 The NVECTOR API

The generic NVECTOR API can be broken down into groups of functions: the core vector operations, the fused vector operations, the vector array operations, the local reduction operations, the exchange operations, and finally some utility functions. All but the last group are defined by a particular NVECTOR implementation. The utility functions are defined by the generic NVECTOR itself.

7.1.1 NVECTOR core functions

`N_VGetVectorID`

Call `id = N_VGetVectorID(w);`

Description Returns the vector type identifier for the vector `w`. It is used to determine the vector implementation type (e.g. serial, parallel, ...) from the abstract `N_Vector` interface.

Arguments `w` (`N_Vector`) a NVECTOR object

Return value This function returns an `N_Vector_ID`. Possible values are given in Table 7.1.

F2003 Name `FN_VGetVectorID`

`N_VClone`

Call `v = N_VClone(w);`

Description Creates a new `N_Vector` of the same type as an existing vector `w` and sets the `ops` field. It does not copy the vector, but rather allocates storage for the new vector.

Arguments `w` (`N_Vector`) a NVECTOR object

Return value This function returns an `N_Vector` object. If an error occurs, then this routine will return `NULL`.

F2003 Name `FN_VClone`

N_VCloneEmpty

Call `v = N_VCloneEmpty(w);`

Description Creates a new `N_Vector` of the same type as an existing vector `w` and sets the `ops` field. It does not allocate storage for data.

Arguments `w` (`N_Vector`) a NVECTOR object

Return value This function returns an `N_Vector` object. If an error occurs, then this routine will return `NULL`.

F2003 Name `FN_VCloneEmpty`

N_VDestroy

Call `N_VDestroy(v);`

Description Destroys the `N_Vector` `v` and frees memory allocated for its internal data.

Arguments `v` (`N_Vector`) a NVECTOR object to destroy

Return value `None`

F2003 Name `FN_VDestroy`

N_VSpace

Call `N_VSpace(v, &lrw, &liw);`

Description Returns storage requirements for one `N_Vector`. `lrw` contains the number of realtype words and `liw` contains the number of integer words. This function is advisory only, for use in determining a user's total space requirements; it could be a dummy function in a user-supplied NVECTOR module if that information is not of interest.

Arguments `v` (`N_Vector`) a NVECTOR object

`lrw` (`sunindextype*`) out parameter containing the number of realtype words

`liw` (`sunindextype*`) out parameter containing the number of integer words

Return value `None`

F2003 Name `FN_VSpace`

F2003 Call `integer(c_long) :: lrw(1), liw(1)`
`call FN_VSpace_Serial(v, lrw, liw)`

N_VGetArrayPointer

Call `vdata = N_VGetArrayPointer(v);`

Description Returns a pointer to a `realtyp` array from the `N_Vector` `v`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realtyp` and is accessible from the CPU.

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.

Arguments `v` (`N_Vector`) a NVECTOR object

Return value `realtyp*`

F2003 Name `FN_VGetArrayPointer`

N_VGetDeviceArrayPointer

Call `vdata = N_VGetDeviceArrayPointer(v);`

Description Returns a device pointer to a `realttype` array from the `N_Vector` `v`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realttype` and is accessible from the device (e.g., GPU).

This operation is *optional* except when using the GPU-enabled direct linear solvers.

Arguments `v` (`N_Vector`) a NVECTOR object

Return value `realttype*`

Notes Currently, only the GPU-enabled SUNDIALS vectors provide this operation. All other SUNDIALS vectors will return `NULL`.

F2003 Name `FN_VGetDeviceArrayPointer`

N_VSetArrayPointer

Call `N_VSetArrayPointer(vdata, v);`

Description Overwrites the pointer to the data in an `N_Vector` with a given `realttype*`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realttype`. This routine is only used in the interfaces to the dense (serial) linear solver, hence need not exist in a user-supplied NVECTOR module for a parallel environment.

Arguments `v` (`N_Vector`) a NVECTOR object

Return value `None`

F2003 Name `FN_VSetArrayPointer`

N_VGetCommunicator

Call `N_VGetCommunicator(v);`

Description Returns a pointer to the `MPI_Comm` object associated with the vector (if applicable). For MPI-unaware vector implementations, this should return `NULL`.

Arguments `v` (`N_Vector`) a NVECTOR object

Return value A `void *` pointer to the `MPI_Comm` object if the vector is MPI-aware, otherwise `NULL`.

F2003 Name `FN_VGetCommunicator`

N_VGetLength

Call `N_VGetLength(v);`

Description Returns the global length (number of ‘active’ entries) in the NVECTOR `v`. This value should be cumulative across all processes if the vector is used in a parallel environment. If `v` contains additional storage, e.g., for parallel communication, those entries should not be included.

Arguments `v` (`N_Vector`) a NVECTOR object

Return value `sunindextype`

F2003 Name `FN_VGetLength`

N_VLinearSum

Call `N_VLinearSum(a, x, b, y, z);`

Description Performs the operation $z = ax + by$, where a and b are **realtype** scalars and x and y are of type **N_Vector**: $z_i = ax_i + by_i$, $i = 0, \dots, n - 1$.

Arguments **a** (**realtype**) constant that scales **x**
x (**N_Vector**) a NVECTOR object
b (**realtype**) constant that scales **y**
y (**N_Vector**) a NVECTOR object
z (**N_Vector**) a NVECTOR object containing the result

Return value The output vector **z** can be the same as either of the input vectors (**x** or **y**).

F2003 Name `FN_VLinearSum`

N_VConst

Call `N_VConst(c, z);`

Description Sets all components of the **N_Vector** **z** to **realtype** **c**: $z_i = c$, $i = 0, \dots, n - 1$.

Arguments **c** (**realtype**) constant to set all components of **z** to
z (**N_Vector**) a NVECTOR object containing the result

Return value None

F2003 Name `FN_VConst`

N_VProd

Call `N_VProd(x, y, z);`

Description Sets the **N_Vector** **z** to be the component-wise product of the **N_Vector** inputs **x** and **y**: $z_i = x_i y_i$, $i = 0, \dots, n - 1$.

Arguments **x** (**N_Vector**) a NVECTOR object
y (**N_Vector**) a NVECTOR object
z (**N_Vector**) a NVECTOR object containing the result

Return value None

F2003 Name `FN_VProd`

N_VDiv

Call `N_VDiv(x, y, z);`

Description Sets the **N_Vector** **z** to be the component-wise ratio of the **N_Vector** inputs **x** and **y**: $z_i = x_i / y_i$, $i = 0, \dots, n - 1$. 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.

Arguments **x** (**N_Vector**) a NVECTOR object
y (**N_Vector**) a NVECTOR object
z (**N_Vector**) a NVECTOR object containing the result

Return value None

F2003 Name `FN_VDiv`

N_VScale

Call `N_VScale(c, x, z);`

Description Scales the `N_Vector` `x` by the `realtype` scalar `c` and returns the result in `z`: $z_i = cx_i, i = 0, \dots, n - 1$.

Arguments `c` (`realtype`) constant that scales the vector `x`
`x` (`N_Vector`) a NVECTOR object
`z` (`N_Vector`) a NVECTOR object containing the result

Return value None

F2003 Name `FN_VScale`

N_VAbs

Call `N_VAbs(x, z);`

Description Sets the components of the `N_Vector` `z` to be the absolute values of the components of the `N_Vector` `x`: $z_i = |x_i|, i = 0, \dots, n - 1$.

Arguments `x` (`N_Vector`) a NVECTOR object
`z` (`N_Vector`) a NVECTOR object containing the result

Return value None

F2003 Name `FN_VAbs`

N_VInv

Call `N_VInv(x, z);`

Description Sets the components of the `N_Vector` `z` to be the inverses of the components of the `N_Vector` `x`: $z_i = 1.0/x_i, i = 0, \dots, n - 1$. 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.

Arguments `x` (`N_Vector`) a NVECTOR object to
`z` (`N_Vector`) a NVECTOR object containing the result

Return value None

F2003 Name `FN_VInv`

N_VAddConst

Call `N_VAddConst(x, b, z);`

Description Adds the `realtype` scalar `b` to all components of `x` and returns the result in the `N_Vector` `z`: $z_i = x_i + b, i = 0, \dots, n - 1$.

Arguments `x` (`N_Vector`) a NVECTOR object
`b` (`realtype`) constant added to all components of `x`
`z` (`N_Vector`) a NVECTOR object containing the result

Return value None

F2003 Name `FN_VAddConst`

N_VDotProd

Call `d = N_VDotProd(x, y);`

Description Returns the value of the ordinary dot product of `x` and `y`: $d = \sum_{i=0}^{n-1} x_i y_i$.

Arguments `x` (`N_Vector`) a NVECTOR object with `y`
`y` (`N_Vector`) a NVECTOR object with `x`

Return value **realtype**

F2003 Name **FN_VDotProd**

N_VMaxNorm

Call $m = \text{N_VMaxNorm}(x);$

Description Returns the maximum norm of the **N_Vector** **x**: $m = \max_i |x_i|$.

Arguments **x** (**N_Vector**) a NVECTOR object

Return value **realtype**

F2003 Name **FN_VMaxNorm**

N_VWrmsNorm

Call $m = \text{N_VWrmsNorm}(x, w)$

Description Returns the weighted root-mean-square norm of the **N_Vector** **x** with **realtype** weight vector **w**: $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i)^2\right) / n}$.

Arguments **x** (**N_Vector**) a NVECTOR object

w (**N_Vector**) a NVECTOR object containing weights

Return value **realtype**

F2003 Name **FN_VWrmsNorm**

N_VWrmsNormMask

Call $m = \text{N_VWrmsNormMask}(x, w, id);$

Description Returns the weighted root mean square norm of the **N_Vector** **x** with **realtype** weight vector **w** built using only the elements of **x** corresponding to positive elements of the

N_Vector **id**: $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i H(id_i))^2\right) / n}$, where $H(\alpha) = \begin{cases} 1 & \alpha > 0 \\ 0 & \alpha \leq 0 \end{cases}$

Arguments **x** (**N_Vector**) a NVECTOR object

w (**N_Vector**) a NVECTOR object containing weights

id (**N_Vector**) mask vector

Return value **realtype**

F2003 Name **FN_VWrmsNormMask**

N_VMin

Call $m = \text{N_VMin}(x);$

Description Returns the smallest element of the **N_Vector** **x**: $m = \min_i x_i$.

Arguments **x** (**N_Vector**) a NVECTOR object

Return value **realtype**

F2003 Name **FN_VMin**

N_VWL2Norm

Call `m = N_VWL2Norm(x, w);`

Description Returns the weighted Euclidean ℓ_2 norm of the `N_Vector` `x` with `realttype` weight vector `w`: $m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2}$.

Arguments `x` (`N_Vector`) a NVECTOR object
`w` (`N_Vector`) a NVECTOR object containing weights

Return value `realttype`

F2003 Name `FN_VWL2Norm`

N_VL1Norm

Call `m = N_VL1Norm(x);`

Description Returns the ℓ_1 norm of the `N_Vector` `x`: $m = \sum_{i=0}^{n-1} |x_i|$.

Arguments `x` (`N_Vector`) a NVECTOR object to obtain the norm of

Return value `realttype`

F2003 Name `FN_VL1Norm`

N_VCompare

Call `N_VCompare(c, x, z);`

Description Compares the components of the `N_Vector` `x` to the `realttype` scalar `c` and returns an `N_Vector` `z` such that: $z_i = 1.0$ if $|x_i| \geq c$ and $z_i = 0.0$ otherwise.

Arguments `c` (`realttype`) constant that each component of `x` is compared to
`x` (`N_Vector`) a NVECTOR object
`z` (`N_Vector`) a NVECTOR object containing the result

Return value `None`

F2003 Name `FN_VCompare`

N_VInvTest

Call `t = N_VInvTest(x, z);`

Description Sets the components of the `N_Vector` `z` to be the inverses of the components of the `N_Vector` `x`, with prior testing for zero values: $z_i = 1.0/x_i$, $i = 0, \dots, n-1$.

Arguments `x` (`N_Vector`) a NVECTOR object
`z` (`N_Vector`) an output NVECTOR object

Return value Returns a `booleantype` with value `SUNTRUE` if all components of `x` are nonzero (successful inversion) and returns `SUNFALSE` otherwise.

F2003 Name `FN_VInvTest`

N_VConstrMask

Call `t = N_VConstrMask(c, x, m);`

Description Performs the following constraint tests: $x_i > 0$ if $c_i = 2$, $x_i \geq 0$ if $c_i = 1$, $x_i \leq 0$ if $c_i = -1$, $x_i < 0$ if $c_i = -2$. There is no constraint on x_i if $c_i = 0$. This routine returns a boolean assigned to `SUNFALSE` if any element failed the constraint test and assigned to `SUNTRUE` if all passed. It also sets a mask vector `m`, with elements equal to 1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking.

Arguments `c` (`realtype`) scalar constraint value
`x` (`N_Vector`) a NVECTOR object
`m` (`N_Vector`) output mask vector

Return value Returns a `booleantype` with value `SUNFALSE` if any element failed the constraint test, and `SUNTRUE` if all passed.

F2003 Name `FN_VConstrMask`

`N_VMinQuotient`

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

Description This routine returns the minimum of the quotients obtained by term-wise dividing `numi` by `denomi`. 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.

Arguments `num` (`N_Vector`) a NVECTOR object used as the numerator
`denom` (`N_Vector`) a NVECTOR object used as the denominator

Return value `realtype`

F2003 Name `FN_VMinQuotient`

7.1.2 NVECTOR fused functions

Fused and vector array 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 a fused or vector array operation as `NULL`, the generic NVECTOR module will automatically call standard vector operations as necessary to complete the desired operation. In all SUNDIALS-provided NVECTOR implementations, all fused and vector array operations are disabled by default. However, these implementations provide additional user-callable functions to enable/disable any or all of the fused and vector array operations. See the following sections for the implementation specific functions to enable/disable operations.

`N_VLinearCombination`

Call `ier = N_VLinearCombination(nv, c, X, z);`

Description This routine computes the linear combination of n_v vectors with n elements:

$$z_i = \sum_{j=0}^{n_v-1} c_j x_{j,i}, \quad i = 0, \dots, n-1,$$

where c is an array of n_v scalars, X is an array of n_v vectors, and z is the output vector.

Arguments `nv` (`int`) the number of vectors in the linear combination
`c` (`realtype*`) an array of n_v scalars used to scale the corresponding vector in X
`X` (`N_Vector*`) an array of n_v NVECTOR objects to be scaled and combined
`z` (`N_Vector`) a NVECTOR object containing the result

Return value Returns an `int` with value 0 for success and a non-zero value otherwise.

Notes If the output vector z is one of the vectors in X , then it *must* be the first vector in the vector array.

F2003 Name `FN_VLinearCombination`

F2003 Call `real(c_double) :: c(nv)`
`type(c_ptr), target :: X(nv)`
`type(N_Vector), pointer :: z`
`ier = FN_VLinearCombination(nv, c, X, z)`

N_VScaleAddMulti

Call `ier = N_VScaleAddMulti(nv, c, x, Y, Z);`

Description This routine scales and adds one vector to n_v vectors with n elements:

$$z_{j,i} = c_j x_i + y_{j,i}, \quad j = 0, \dots, n_v - 1 \quad i = 0, \dots, n - 1,$$

where c is an array of n_v scalars, x is the vector to be scaled and added to each vector in the vector array of n_v vectors Y , and Z is a vector array of n_v output vectors.

Arguments `nv` (`int`) the number of scalars and vectors in `c`, `Y`, and `Z`
`c` (`realtype*`) an array of n_v scalars
`x` (`N_Vector`) a NVECTOR object to be scaled and added to each vector in `Y`
`Y` (`N_Vector*`) an array of n_v NVECTOR objects where each vector j will have the vector `x` scaled by `c_j` added to it
`Z` (`N_Vector`) an output array of n_v NVECTOR objects

Return value Returns an `int` with value 0 for success and a non-zero value otherwise.

F2003 Name `FN_VScaleAddMulti`

F2003 Call `real(c_double) :: c(nv)`
`type(c_ptr), target :: Y(nv), Z(nv)`
`type(N_Vector), pointer :: x`
`ierr = FN_VScaleAddMulti(nv, c, x, Y, Z)`

N_VDotProdMulti

Call `ier = N_VDotProdMulti(nv, x, Y, d);`

Description This routine computes the dot product of a vector with n_v other vectors:

$$d_j = \sum_{i=0}^{n-1} x_i y_{j,i}, \quad j = 0, \dots, n_v - 1,$$

where d is an array of n_v scalars containing the dot products of the vector x with each of the n_v vectors in the vector array Y .

Arguments `nv` (`int`) the number of vectors in `Y`
`x` (`N_Vector`) a NVECTOR object to be used in a dot product with each of the vectors in `Y`
`Y` (`N_Vector*`) an array of n_v NVECTOR objects to use in a dot product with `x`
`d` (`realtype*`) an output array of n_v dot products

Return value Returns an `int` with value 0 for success and a non-zero value otherwise.

F2003 Name `FN_VDotProdMulti`

F2003 Call `real(c_double) :: d(nv)`
`type(c_ptr), target :: Y(nv)`
`type(N_Vector), pointer :: x`
`ierr = FN_VDotProdMulti(nv, x, Y, d)`

7.1.3 NVECTOR vector array functions

N_VLinearSumVectorArray

Call `ier = N_VLinearSumVectorArray(nv, a, X, b, Y, Z);`

Description This routine computes the linear sum of two vector arrays containing n_v vectors of n elements:

$$z_{j,i} = ax_{j,i} + by_{j,i}, \quad i = 0, \dots, n-1 \quad j = 0, \dots, n_v-1,$$

where a and b are scalars and X , Y , and Z are arrays of n_v vectors.

Arguments `nv` (`int`) the number of vectors in the vector arrays
`a` (`realtype`) constant to scale each vector in X by
`X` (`N_Vector*`) an array of n_v NVECTOR objects
`Y` (`N_Vector*`) an array of n_v NVECTOR objects
`Z` (`N_Vector*`) an output array of n_v NVECTOR objects

Return value Returns an `int` with value 0 for success and a non-zero value otherwise.

F2003 Name `FN_VLinearSumVectorArray`

N_VScaleVectorArray

Call `ier = N_VScaleVectorArray(nv, c, X, Z);`

Description This routine scales each vector of n elements in a vector array of n_v vectors by a potentially different constant:

$$z_{j,i} = c_j x_{j,i}, \quad i = 0, \dots, n-1 \quad j = 0, \dots, n_v-1,$$

where c is an array of n_v scalars and X and Z are arrays of n_v vectors.

Arguments `nv` (`int`) the number of vectors in the vector arrays
`c` (`realtype`) constant to scale each vector in X by
`X` (`N_Vector*`) an array of n_v NVECTOR objects
`Z` (`N_Vector*`) an output array of n_v NVECTOR objects

Return value Returns an `int` with value 0 for success and a non-zero value otherwise.

F2003 Name `FN_VScaleVectorArray`

N_VConstVectorArray

Call `ier = N_VConstVectorArray(nv, c, X);`

Description This routine sets each element in a vector of n elements in a vector array of n_v vectors to the same value:

$$z_{j,i} = c, \quad i = 0, \dots, n-1 \quad j = 0, \dots, n_v-1,$$

where c is a scalar and X is an array of n_v vectors.

Arguments `nv` (`int`) the number of vectors in X
`c` (`realtype`) constant to set every element in every vector of X to
`X` (`N_Vector*`) an array of n_v NVECTOR objects

Return value Returns an `int` with value 0 for success and a non-zero value otherwise.

F2003 Name `FN_VConstVectorArray`

N_VWrmsNormVectorArray

Call `ier = N_VWrmsNormVectorArray(nv, X, W, m);`

Description This routine computes the weighted root mean square norm of n_v vectors with n elements:

$$m_j = \left(\frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i} w_{j,i})^2 \right)^{1/2}, \quad j = 0, \dots, n_v - 1,$$

where m contains the n_v norms of the vectors in the vector array X with corresponding weight vectors W .

Arguments **nv** (**int**) the number of vectors in the vector arrays
X (**N_Vector***) an array of n_v NVECTOR objects
W (**N_Vector***) an array of n_v NVECTOR objects
m (**realtype***) an output array of n_v norms

Return value Returns an **int** with value 0 for success and a non-zero value otherwise.

F2003 Name FN_VWrmsNormVectorArray

N_VWrmsNormMaskVectorArray

Call `ier = N_VWrmsNormMaskVectorArray(nv, X, W, id, m);`

Description This routine computes the masked weighted root mean square norm of n_v vectors with n elements:

$$m_j = \left(\frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i} w_{j,i} H(id_i))^2 \right)^{1/2}, \quad j = 0, \dots, n_v - 1,$$

$H(id_i) = 1$ for $id_i > 0$ and is zero otherwise, m contains the n_v norms of the vectors in the vector array X with corresponding weight vectors W and mask vector id .

Arguments **nv** (**int**) the number of vectors in the vector arrays
X (**N_Vector***) an array of n_v NVECTOR objects
W (**N_Vector***) an array of n_v NVECTOR objects
id (**N_Vector**) the mask vector
m (**realtype***) an output array of n_v norms

Return value Returns an **int** with value 0 for success and a non-zero value otherwise.

F2003 Name FN_VWrmsNormMaskVectorArray

N_VScaleAddMultiVectorArray

Call `ier = N_VScaleAddMultiVectorArray(nv, ns, c, X, YY, ZZ);`

Description This routine scales and adds a vector in a vector array of n_v vectors to the corresponding vector in n_s vector arrays:

$$z_{k,j,i} = c_k x_{j,i} + y_{k,j,i}, \quad i = 0, \dots, n - 1 \quad j = 0, \dots, n_v - 1, \quad k = 0, \dots, n_s - 1$$

where c is an array of n_s scalars, X is a vector array of n_v vectors to be scaled and added to the corresponding vector in each of the n_s vector arrays in the array of vector arrays YY and stored in the output array of vector arrays ZZ .

Arguments **nv** (**int**) the number of vectors in the vector arrays
ns (**int**) the number of scalars in c and vector arrays in YY and ZZ
c (**realtype***) an array of n_s scalars

X (**N_Vector***) an array of n_v NVECTOR objects
 YY (**N_Vector****) an array of n_s NVECTOR arrays
 ZZ (**N_Vector****) an output array of n_s NVECTOR arrays

Return value Returns an **int** with value 0 for success and a non-zero value otherwise.

N_VLinearCombinationVectorArray

Call `ier = N_VLinearCombinationVectorArray(nv, ns, c, XX, Z);`

Description This routine computes the linear combination of n_s vector arrays containing n_v vectors with n elements:

$$z_{j,i} = \sum_{k=0}^{n_s-1} c_k x_{k,j,i}, \quad i = 0, \dots, n-1 \quad j = 0, \dots, n_v-1,$$

where c is an array of n_s scalars (type **realtype***), XX (type **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 **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 .

Arguments **nv** (**int**) the number of vectors in the vector arrays
ns (**int**) the number of scalars in c and vector arrays in YY and ZZ
c (**realtype***) an array of n_s scalars
XX (**N_Vector****) an array of n_s NVECTOR arrays
Z (**N_Vector***) an output array NVECTOR objects

Return value Returns an **int** with value 0 for success and a non-zero value otherwise.

7.1.4 NVECTOR local reduction functions

Local reduction operations are intended to reduce parallel communication on distributed memory systems, particularly when NVECTOR objects are combined together within a NVECTOR_MPIMANYVECTOR object (see Section 7.16). If a particular NVECTOR implementation defines a local reduction operation as **NULL**, the NVECTOR_MPIMANYVECTOR module will automatically call standard vector reduction operations as necessary to complete the desired operation. All SUNDIALS-provided NVECTOR implementations include these local reduction operations, which may be used as templates for user-defined NVECTOR implementations.

N_VDotProdLocal

Call `d = N_VDotProdLocal(x, y);`

Description This routine computes the MPI task-local portion of the ordinary dot product of \mathbf{x} and \mathbf{y} :

$$d = \sum_{i=0}^{n_{local}-1} x_i y_i,$$

where n_{local} corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications).

Arguments **x** (**N_Vector**) a NVECTOR object
y (**N_Vector**) a NVECTOR object

Return value **realtype**

F2003 Name **FN_VDotProdLocal**

N_VMaxNormLocal

Call `m = N_VMaxNormLocal(x);`

Description This routine computes the MPI task-local portion of the maximum norm of the `N_Vector` `x`:

$$m = \max_{0 \leq i < n_{local}} |x_i|,$$

where n_{local} corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications).

Arguments `x` (`N_Vector`) a `NVECTOR` object

Return value `realtype`

F2003 Name `FN_VMaxNormLocal`

N_VMinLocal

Call `m = N_VMinLocal(x);`

Description This routine computes the smallest element of the MPI task-local portion of the `N_Vector` `x`:

$$m = \min_{0 \leq i < n_{local}} x_i,$$

where n_{local} corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications).

Arguments `x` (`N_Vector`) a `NVECTOR` object

Return value `realtype`

F2003 Name `FN_VMinLocal`

N_VL1NormLocal

Call `n = N_VL1NormLocal(x);`

Description This routine computes the MPI task-local portion of the ℓ_1 norm of the `N_Vector` `x`:

$$n = \sum_{i=0}^{n_{local}-1} |x_i|,$$

where n_{local} corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications).

Arguments `x` (`N_Vector`) a `NVECTOR` object

Return value `realtype`

F2003 Name `FN_VL1NormLocal`

N_VWSqrSumLocal

Call `s = N_VWSqrSumLocal(x,w);`

Description This routine computes the MPI task-local portion of the weighted squared sum of the `N_Vector` `x` with weight vector `w`:

$$s = \sum_{i=0}^{n_{local}-1} (x_i w_i)^2,$$

where n_{local} corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications).

Arguments **x** (`N_Vector`) a NVECTOR object
w (`N_Vector`) a NVECTOR object containing weights

Return value `realtype`

F2003 Name `FN_VWSqrSumLocal`

`N_VWSqrSumMaskLocal`

Call `s = N_VWSqrSumMaskLocal(x,w,id);`

Description This routine computes the MPI task-local portion of the weighted squared sum of the `N_Vector` **x** with weight vector **w** built using only the elements of **x** corresponding to positive elements of the `N_Vector` **id**:

$$m = \sum_{i=0}^{n_{local}-1} (x_i w_i H(id_i))^2, \quad \text{where } H(\alpha) = \begin{cases} 1 & \alpha > 0 \\ 0 & \alpha \leq 0 \end{cases}$$

and n_{local} corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications).

Arguments **x** (`N_Vector`) a NVECTOR object
w (`N_Vector`) a NVECTOR object containing weights
id (`N_Vector`) a NVECTOR object used as a mask

Return value `realtype`

F2003 Name `FN_VWSqrSumMaskLocal`

`N_VInvTestLocal`

Call `t = N_VInvTestLocal(x, z);`

Description Sets the MPI task-local components of the `N_Vector` **z** to be the inverses of the components of the `N_Vector` **x**, with prior testing for zero values:

$$z_i = 1.0/x_i, \quad i = 0, \dots, n_{local} - 1,$$

where n_{local} corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications).

Arguments **x** (`N_Vector`) a NVECTOR object
z (`N_Vector`) an output NVECTOR object

Return value Returns a `booleantype` with the value `SUNTRUE` if all task-local components of **x** are nonzero (successful inversion) and with the value `SUNFALSE` otherwise.

F2003 Name `FN_VInvTestLocal`

`N_VConstrMaskLocal`

Call `t = N_VConstrMaskLocal(c,x,m);`

Description Performs the following constraint tests:

$$\begin{aligned} x_i > 0 & \text{ if } c_i = 2, \\ x_i \geq 0 & \text{ if } c_i = 1, \\ x_i \leq 0 & \text{ if } c_i = -1, \\ x_i < 0 & \text{ if } c_i = -2, \text{ and} \\ \text{no test} & \text{ if } c_i = 0, \end{aligned}$$

for all MPI task-local components of the vectors. It 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.

Arguments `c` (`realtype`) scalar constraint value
`x` (`N_Vector`) a NVECTOR object
`m` (`N_Vector`) output mask vector

Return value Returns a `booleantype` with the value `SUNFALSE` if any task-local element failed the constraint test and the value `SUNTRUE` if all passed.

F2003 Name `FN_VConstrMaskLocal`

N_VMinQuotientLocal

Call `minq = N_VMinQuotientLocal(num,denom);`

Description This routine returns the minimum of the quotients obtained by term-wise dividing `numi` by `denomi`, for all MPI task-local components of the vectors. 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.

Arguments `num` (`N_Vector`) a NVECTOR object used as the numerator
`denom` (`N_Vector`) a NVECTOR object used as the denominator

Return value `realtype`

F2003 Name `FN_VMinQuotientLocal`

7.1.5 NVECTOR exchange operations

The following vector exchange operations are also *optional* and are intended only for use when interfacing with the XBraid library for parallel-in-time integration. In that setting these operations are required but are otherwise unused by SUNDIALS packages and may be set to `NULL`. For each operation, we give the function signature, a description of the expected behavior, and an example of the function usage.

N_VBufSize

Call `flag = N_VBufSize(N_Vector x, sunindextype *size);`

Description This routine returns the buffer size need to exchange in the data in the vector `x` between computational nodes.

Arguments `x` (`N_Vector`) a NVECTOR object
`size` (`sunindextype*`) the size of the message buffer

Return value Returns an `int` with value 0 for success and a non-zero value otherwise.

F2003 Name `FN_VBufSize`

N_VBufPack

Call `flag = N_VBufPack(N_Vector x, void *buf);`

Description This routine fills the exchange buffer `buf` with the vector data in `x`.

Arguments `x` (`N_Vector`) a NVECTOR object
`buf` (`sunindextype*`) the exchange buffer to pack

Return value Returns an `int` with value 0 for success and a non-zero value otherwise.

F2003 Name `FN_VBufPack`

N_VBufUnpack

Call `flag = N_VBufUnpack(N_Vector x, void *buf);`

Description This routine unpacks the data in the exchange buffer `buf` into the vector `x`.

Arguments `x` (`N_Vector`) a NVECTOR object
`buf` (`sunindextype*`) the exchange buffer to unpack

Return value Returns an `int` with value 0 for success and a non-zero value otherwise.

F2003 Name `FN_VBufUnpack`

7.1.6 NVECTOR utility functions

To aid in the creation of custom NVECTOR modules the generic NVECTOR module provides three utility functions `N_VNewEmpty`, `N_VCopyOps` and `N_VFreeEmpty`. When used in custom NVECTOR constructors and clone routines these functions will ease the introduction of any new optional vector operations to the NVECTOR API by ensuring only required operations need to be set and all operations are copied when cloning a vector.

To aid the use of arrays of NVECTOR objects, the generic NVECTOR module also provides the utility functions `N_VCloneVectorArray`, `N_VCloneVectorArrayEmpty`, and `N_VDestroyVectorArray`.

N_VNewEmpty

Call `v = N_VNewEmpty();`

Description The function `N_VNewEmpty` allocates a new generic NVECTOR object and initializes its content pointer and the function pointers in the operations structure to `NULL`.

Arguments None

Return value This function returns an `N_Vector` object. If an error occurs when allocating the object, then this routine will return `NULL`.

F2003 Name `FN_VNewEmpty`

N_VCopyOps

Call `retval = N_VCopyOps(w, v);`

Description The function `N_VCopyOps` copies the function pointers in the `ops` structure of `w` into the `ops` structure of `v`.

Arguments `w` (`N_Vector`) the vector to copy operations from
`v` (`N_Vector`) the vector to copy operations to

Return value This returns 0 if successful and a non-zero value if either of the inputs are `NULL` or the `ops` structure of either input is `NULL`.

F2003 Name `FN_VCopyOps`

N_VFreeEmpty

Call `N_VFreeEmpty(v);`

Description This routine frees the generic `N_Vector` object, under the assumption that any implementation-specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the `ops` pointer is `NULL`, and, if it is not, it will free it as well.

Arguments `v` (`N_Vector`)

Return value None

F2003 Name `FN_VFreeEmpty`

N_VCloneEmptyVectorArray

Call `vecarray = N_VCloneEmptyVectorArray(count, w);`

Description Creates an array of `count` variables of type `N_Vector`, each of the same type as the existing `N_Vector` `w`. It achieves this by calling the implementation-specific `N_VCloneEmpty` operation.

Arguments `count` (`int`) the size of the vector array
`w` (`N_Vector`) the vector to clone

Return value Returns an array of `count` `N_Vector` objects if successful, or `NULL` if an error occurred while cloning.

N_VCloneVectorArray

Call `vecarray = N_VCloneVectorArray(count, w);`

Description Creates an array of `count` variables of type `N_Vector`, each of the same type as the existing `N_Vector` `w`. It achieves this by calling the implementation-specific `N_VClone` operation.

Arguments `count` (`int`) the size of the vector array
`w` (`N_Vector`) the vector to clone

Return value Returns an array of `count` `N_Vector` objects if successful, or `NULL` if an error occurred while cloning.

N_VDestroyVectorArray

Call `N_VDestroyVectorArray(count, w);`

Description Destroys (frees) an array of variables of type `N_Vector`. It depends on the implementation-specific `N_VDestroy` operation.

Arguments `vs` (`N_Vector*`) the array of vectors to destroy
`count` (`int`) the size of the vector array

Return value None

N_VNewVectorArray

Call `vecarray = N_VNewVectorArray(count);`

Description Returns an empty `N_Vector` array large enough to hold `count` `N_Vector` objects. This function is primarily meant for users of the Fortran 2003 interface.

Arguments `count` (`int`) the size of the vector array

Return value Returns a `N_Vector*` if successful, Returns `NULL` if an error occurred.

Notes Users of the Fortran 2003 interface to the `N_VManyVector` or `N_VMPIManyVector` will need this to create an array to hold the subvectors. Note that this function does restrict the the max number of subvectors usable with the `N_VManyVector` and `N_VMPIManyVector` to the max size of an `int` despite the `ManyVector` implementations accepting a subvector count larger than this value.

F2003 Name `FN_VNewVectorArray`

Table 7.1: Vector Identifications associated with vector kernels supplied with SUNDIALS.

Vector ID	Vector type	ID Value
SUNDIALS_NVEC_SERIAL	Serial	0
SUNDIALS_NVEC_PARALLEL	Distributed memory parallel (MPI)	1
SUNDIALS_NVEC_OPENMP	OpenMP shared memory parallel	2
SUNDIALS_NVEC_PTHREADS	PThreads shared memory parallel	3
SUNDIALS_NVEC_PARHYP	<i>hypr</i> ParHyp parallel vector	4
SUNDIALS_NVEC_PETSC	PETSc parallel vector	5
SUNDIALS_NVEC_CUDA	CUDA vector	6
SUNDIALS_NVEC_HIP	HIP vector	7
SUNDIALS_NVEC_SYCL	SYCL vector	8
SUNDIALS_NVEC_RAJA	RAJA vector	9
SUNDIALS_NVEC_OPENMPDEV	OpenMP vector with device offloading	10
SUNDIALS_NVEC_TRILINOS	Trilinos Tpetra vector	11
SUNDIALS_NVEC_MANYVECTOR	“ManyVector” vector	12
SUNDIALS_NVEC_MPIMANYVECTOR	MPI-enabled “ManyVector” vector	13
SUNDIALS_NVEC_MPIPLUSX	MPI+X vector	14
SUNDIALS_NVEC_CUSTOM	User-provided custom vector	15

N_VGetVecAtIndexVectorArray

Call `v = N_VGetVecAtIndexVectorArray(vecs, index);`

Description Returns the `N_Vector` object stored in the vector array at the provided index. This function is primarily meant for users of the Fortran 2003 interface.

Arguments `vecs` (`N_Vector*`) the array of vectors to index
`index` (`int`) the index of the vector to return

Return value Returns the `N_Vector` object stored in the vector array at the provided index. Returns `NULL` if an error occurred.

F2003 Name `FN_VGetVecAtIndexVectorArray`

N_VSetVecAtIndexVectorArray

Call `N_VSetVecAtIndexVectorArray(vecs, index, v);`

Description Sets the `N_Vector` object stored in the vector array at the provided index. This function is primarily meant for users of the Fortran 2003 interface.

Arguments `vecs` (`N_Vector*`) the array of vectors to index
`index` (`int`) the index of the vector to return
`v` (`N_Vector`) the vector to store at the index

Return value None

F2003 Name `FN_VSetVecAtIndexVectorArray`

7.1.7 NVECTOR identifiers

Each `NVECTOR` implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 7.1.

7.1.8 The generic NVECTOR module implementation

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);
    realtype* (*nvgetdevicearraypointer)(N_Vector);
    void (*nvsetarraypointer)(realtype *, N_Vector);
    void* (*nvgetcommunicator)(N_Vector);
    sunindextype (*nvgetlength)(N_Vector);
    void (*nvlinearsum)(realtype, N_Vector, realtype, N_Vector, N_Vector);
    void (*nvconst)(realtype, N_Vector);
    void (*nvprod)(N_Vector, N_Vector, N_Vector);
    void (*nvdiv)(N_Vector, N_Vector, N_Vector);
    void (*nvscale)(realtype, N_Vector, N_Vector);
    void (*nvabs)(N_Vector, N_Vector);
    void (*nvinv)(N_Vector, N_Vector);
    void (*nvaddconst)(N_Vector, realtype, N_Vector);
    realtype (*nvdotprod)(N_Vector, N_Vector);
    realtype (*nvmaxnorm)(N_Vector);
    realtype (*nvwrmsnorm)(N_Vector, N_Vector);
    realtype (*nvwrmsnormmask)(N_Vector, N_Vector, N_Vector);
    realtype (*nvmin)(N_Vector);
    realtype (*nvwl2norm)(N_Vector, N_Vector);
    realtype (*nvl1norm)(N_Vector);
    void (*nvcompare)(realtype, N_Vector, N_Vector);
    booleantype (*nvinvtest)(N_Vector, N_Vector);
    booleantype (*nvconstrmask)(N_Vector, N_Vector, N_Vector);
    realtype (*nvminquotient)(N_Vector, N_Vector);
    int (*nvlinearcombination)(int, realtype*, N_Vector*, N_Vector);
    int (*nvscaleaddmulti)(int, realtype*, N_Vector, N_Vector*, N_Vector*);
    int (*nvdotprodmulti)(int, N_Vector, N_Vector*, realtype*);
    int (*nvlinearsumvectorarray)(int, realtype, N_Vector*, realtype,
                                  N_Vector*, N_Vector*);
    int (*nvscalevectorarray)(int, realtype*, N_Vector*, N_Vector*);
    int (*nvconstvectorarray)(int, realtype, N_Vector*);
    int (*nvwrmsnomrvectorarray)(int, N_Vector*, N_Vector*, realtype*);
    int (*nvwrmsnomrmaskvectorarray)(int, N_Vector*, N_Vector*, N_Vector,
                                      realtype*);
    int (*nvscaleaddmultivectorarray)(int, int, realtype*, N_Vector*,
                                       N_Vector**, N_Vector**);
    int (*nvlinearcombinationvectorarray)(int, int, realtype*, N_Vector**,
                                           N_Vector*);
    realtype (*nvdotprodlocal)(N_Vector, N_Vector);
```

```

realtype      (*nvmaxnormlocal)(N_Vector);
realtype      (*nvminlocal)(N_Vector);
realtype      (*nvlnormlocal)(N_Vector);
boolean_type  (*nvinvtestlocal)(N_Vector, N_Vector);
boolean_type  (*nvconstrmasklocal)(N_Vector, N_Vector, N_Vector);
realtype      (*nvminquotientlocal)(N_Vector, N_Vector);
realtype      (*nvwsqrsumlocal)(N_Vector, N_Vector);
realtype      (*nvwsqrsummasklocal)(N_Vector, N_Vector, N_Vector);
int           (*nvbufsize)(N_Vector, sunindex_type *);
int           (*nvbufpack)(N_Vector, void*);
int           (*nvbufunpack)(N_Vector, void*);
};

```

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`:

```

void N_VScale(realtype c, N_Vector x, N_Vector z)
{
    z->ops->nvscale(c, x, z);
}

```

Section 7.1.1 defines a complete list of all standard vector operations defined by the generic NVECTOR module. Sections 7.1.2, 7.1.3 and 7.1.4 list *optional* fused, vector array and local reduction operations, respectively.

The Fortran 2003 interface provides a `bind(C)` derived-type for the `_generic_N_Vector` and the `_generic_N_Vector_Ops` structures. Their definition is given below.

```

type, bind(C), public :: N_Vector
  type(C_PTR), public :: content
  type(C_PTR), public :: ops
end type N_Vector

type, bind(C), public :: N_Vector_Ops
  type(C_FUNPTR), public :: nvgetvectorid
  type(C_FUNPTR), public :: nvclone
  type(C_FUNPTR), public :: nvcloneempty
  type(C_FUNPTR), public :: nvdestroy
  type(C_FUNPTR), public :: nvspace
  type(C_FUNPTR), public :: nvgetarraypointer
  type(C_FUNPTR), public :: nvsetarraypointer
  type(C_FUNPTR), public :: nvgetcommunicator
  type(C_FUNPTR), public :: nvgetlength
  type(C_FUNPTR), public :: nvlinearsum
  type(C_FUNPTR), public :: nvconst
  type(C_FUNPTR), public :: nvprod
  type(C_FUNPTR), public :: nvdiv
  type(C_FUNPTR), public :: nvscale
  type(C_FUNPTR), public :: nvabs
  type(C_FUNPTR), public :: nvinv
  type(C_FUNPTR), public :: nvaddconst
  type(C_FUNPTR), public :: nvdotprod
  type(C_FUNPTR), public :: nvmaxnorm
  type(C_FUNPTR), public :: nvwrmsnorm

```

```

type(C_FUNPTR), public :: nvrmsnormmask
type(C_FUNPTR), public :: nvmin
type(C_FUNPTR), public :: nwl2norm
type(C_FUNPTR), public :: nvl1norm
type(C_FUNPTR), public :: nvcompare
type(C_FUNPTR), public :: nvinvtest
type(C_FUNPTR), public :: nvconstrmask
type(C_FUNPTR), public :: nvminquotient
type(C_FUNPTR), public :: nvlinearcombination
type(C_FUNPTR), public :: nvscaleaddmulti
type(C_FUNPTR), public :: nvdotprodmulti
type(C_FUNPTR), public :: nvlinearsumvectorarray
type(C_FUNPTR), public :: nvscalevectorarray
type(C_FUNPTR), public :: nvconstvectorarray
type(C_FUNPTR), public :: nvrmsnormvectorarray
type(C_FUNPTR), public :: nvrmsnormmaskvectorarray
type(C_FUNPTR), public :: nvscaleaddmultivecarray
type(C_FUNPTR), public :: nvlinearcombinationvectorarray
type(C_FUNPTR), public :: nvdotprodlocal
type(C_FUNPTR), public :: nvmaxnormlocal
type(C_FUNPTR), public :: nvminlocal
type(C_FUNPTR), public :: nvl1normlocal
type(C_FUNPTR), public :: nvinvtestlocal
type(C_FUNPTR), public :: nvconstrmasklocal
type(C_FUNPTR), public :: nvminquotientlocal
type(C_FUNPTR), public :: nvwsqrsumlocal
type(C_FUNPTR), public :: nvwsqrsummasklocal
type(C_FUNPTR), public :: nvbufsize
type(C_FUNPTR), public :: nvbufpack
type(C_FUNPTR), public :: nvbufunpack
end type N_Vector_Ops

```

7.1.9 Implementing a custom NVECTOR

A particular implementation of the NVECTOR module must:

- Specify the *content* field of `N_Vector`.
- Define and implement the vector operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one NVECTOR module (each with different `N_Vector` internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free an `N_Vector` with the new *content* field and with *ops* pointing to the new vector operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined `N_Vector` (e.g., a routine to print the content for debugging purposes).
- Optionally, provide accessor macros as needed for that particular implementation to be used to access different parts in the *content* field of the newly defined `N_Vector`.

It is recommended that a user-supplied NVECTOR implementation returns the `SUNDIALS_NVEC_CUSTOM` identifier from the `N_VGetVectorID` function.

To aid in the creation of custom NVECTOR modules the generic NVECTOR module provides two utility functions `N_VNewEmpty` and `N_VCopyOps`. When used in custom NVECTOR constructors and clone routines these functions will ease the introduction of any new optional vector operations to the

NVECTOR API by ensuring only required operations need to be set and all operations are copied when cloning a vector.

7.1.9.1 Support for complex-valued vectors

While SUNDIALS itself is written under an assumption of real-valued data, it does provide limited support for complex-valued problems. However, since none of the built-in NVECTOR modules supports complex-valued data, users must provide a custom NVECTOR implementation for this task. Many of the NVECTOR routines described in Sections 7.1.1-7.1.4 above naturally extend to complex-valued vectors; however, some do not. To this end, we provide the following guidance:

- `N_VMin` and `N_VMinLocal` should return the minimum of all *real* components of the vector, i.e., $m = \min_i \text{real}(x_i)$.
- `N_VConst` (and similarly `N_VConstVectorArray`) should set the real components of the vector to the input constant, and set all imaginary components to zero, i.e., $z_i = c + 0j$, $i = 0, \dots, n - 1$.
- `N_VAddConst` should only update the real components of the vector with the input constant, leaving all imaginary components unchanged.
- `N_VWrmsNorm`, `N_VWrmsNormMask`, `N_VWSqrSumLocal` and `N_VWSqrSumMaskLocal` should assume that all entries of the weight vector `w` and the mask vector `id` are real-valued.
- `N_VDotProd` should mathematically return a complex number for complex-valued vectors; as this is not possible with SUNDIALS' current `realtype`, this routine should be set to `NULL` in the custom NVECTOR implementation.
- `N_VCompare`, `N_VConstrMask`, `N_VMinQuotient`, `N_VConstrMaskLocal` and `N_VMinQuotientLocal` are ill-defined due to the lack of a clear ordering in the complex plane. These routines should be set to `NULL` in the custom NVECTOR implementation.

While many SUNDIALS solver modules may be utilized on complex-valued data, others cannot. Specifically, although both `SUNNONLINSOL_NEWTON` and `SUNNONLINSOL_FIXEDPOINT` may be used with any of the IVP solvers (`CVODE`, `CVODES`, `IDA`, `IDAS` and `ARKODE`) for complex-valued problems, the Anderson-acceleration feature `SUNNONLINSOL_FIXEDPOINT` cannot be used due to its reliance on `N_VDotProd`. By this same logic, the Anderson acceleration feature within `KINSOL` also will not work with complex-valued vectors.

Similarly, although each package's linear solver interface (e.g., `CVLS`) may be used on complex-valued problems, none of the built-in `SUNMATRIX` or `SUNLINSOL` modules work. Hence a complex-valued user should provide a custom `SUNLINSOL` (and optionally a custom `SUNMATRIX`) implementation for solving linear systems, and then attach this module as normal to the package's linear solver interface.

Finally, constraint-handling features of each package cannot be used for complex-valued data, due to the issue of ordering in the complex plane discussed above with `N_VCompare`, `N_VConstrMask`, `N_VMinQuotient`, `N_VConstrMaskLocal` and `N_VMinQuotientLocal`.

We provide a simple example of a complex-valued example problem, including a custom complex-valued Fortran 2003 NVECTOR module, in the files `examples/arkode/F2003_custom/ark_analytic_complex_f2003.f90`, `examples/arkode/F2003_custom/fnvector_complex_mod.f90`, and `examples/arkode/F2003_custom/test_fnvector_complex_mod.f90`.

7.2 NVECTOR functions used by KINSOL

In Table 7.2 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 7.2: List of vector functions usage by KINSOL code modules

	KINSOL	KINLS	KINBBDPRE	FKINSOL
N_VGetVectorID				
N_VGetLength		4		
N_VClone	✓		✓	
N_VCloneEmpty				✓
N_VDestroy	✓		✓	✓
N_VSpace	✓	2		
N_VGetArrayPointer		1	✓	✓
N_VSetArrayPointer		1		✓
N_VLinearSum	✓	✓		
N_VConst		✓		
N_VProd	✓	✓		
N_VDiv	✓			
N_VScale	✓	✓	✓	
N_VAbs	✓			
N_VInv	✓			
N_VDotProd	✓	✓		
N_VMaxNorm	✓			
N_VMin	✓			
N_VWL2Norm	✓	✓		
N_VL1Norm		3		
N_VConstrMask	✓			
N_VMinQuotient	✓			
N_VLinearCombination	✓	✓		
N_VDotProdMulti	✓			

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.
4. This routine is only used when an iterative SUNLINSOL module that does not support the `SUNLinSolSetScalingVectors` routine is supplied to KINSOL.

Each SUNLINSOL object may require additional NVECTOR routines not listed in the table above. Please see the the relevant descriptions of these modules in Sections 9.5-9.19 for additional detail on their NVECTOR requirements.

The vector functions listed in Table 7.1.1 that are *not* used by KINSOL are `N_VAddConst`, `N_VWrmsNorm`, `N_VWrmsNormMask`, `N_VCompare`, `N_VInvTest`, and `N_VGetCommunicator`. Therefore a user-supplied NVECTOR module for KINSOL could omit these functions.

The optional function `N_VLinearCombination` is only used when Anderson acceleration is enabled or the SPBCGS, SPTFQMR, SPGMR, or SPFGMR linear solvers are used. `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 7.1.2 and 7.1.3 are unused and a user-supplied NVECTOR module for KINSOL could omit these operations.

7.3 The NVECTOR_SERIAL implementation

The serial implementation of the NVECTOR module provided with SUNDIALS, `NVECTOR_SERIAL`, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, and a boolean flag *own_data* which specifies the ownership of *data*.

```
struct _N_VectorContent_Serial {
    sunindextype length;
    booleantype own_data;
    realtype *data;
};
```

The header file to include when using this module is `nvector_serial.h`. The installed module library to link to is `libsundials_nvecserial.lib` where *.lib* is typically *.so* for shared libraries and *.a* for static libraries.

7.3.1 NVECTOR_SERIAL accessor macros

The following macros are provided to access the content of an `NVECTOR_SERIAL` vector. The suffix `_S` in the names denotes the serial version.

- `NV_CONTENT_S`

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

The assignment `v_cont = NV_CONTENT_S(v)` sets `v_cont` to be a pointer to the serial `N_Vector` content structure.

Implementation:

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

- `NV_OWN_DATA_S`, `NV_DATA_S`, `NV_LENGTH_S`

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

The assignment `v_data = NV_DATA_S(v)` sets `v_data` to be a pointer to the first component of the data for the `N_Vector` `v`. The assignment `NV_DATA_S(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

The assignment `v_len = NV_LENGTH_S(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_S(v) = len_v` sets the length of `v` to be `len_v`.

Implementation:

```
#define NV_OWN_DATA_S(v) ( NV_CONTENT_S(v)->own_data )
#define NV_DATA_S(v) ( NV_CONTENT_S(v)->data )
#define NV_LENGTH_S(v) ( NV_CONTENT_S(v)->length )
```

- NV_Ith_S

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

The assignment `r = NV_Ith_S(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_S(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

Here `i` ranges from 0 to `n - 1` for a vector of length `n`.

Implementation:

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

7.3.2 NVECTOR_SERIAL functions

The `NVECTOR_SERIAL` module defines serial implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3 and 7.1.4. Their names are obtained from those in these tables by appending the suffix `_Serial` (e.g. `N_VDestroy_Serial`). All the standard vector operations listed in 7.1.1 with the suffix `_Serial` appended are callable via the FORTRAN 2003 interface by prepending an 'F' (e.g. `FN_VDestroy_Serial`).

The module `NVECTOR_SERIAL` provides the following additional user-callable routines:

N_VNew_Serial

Prototype `N_Vector N_VNew_Serial(sunindextype vec_length);`

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

F2003 Name This function is callable as `FN_VNew_Serial` when using the Fortran 2003 interface module.

N_VNewEmpty_Serial

Prototype `N_Vector N_VNewEmpty_Serial(sunindextype vec_length);`

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

F2003 Name This function is callable as `FN_VNewEmpty_Serial` when using the Fortran 2003 interface module.

N_VMake_Serial

Prototype `N_Vector N_VMake_Serial(sunindextype vec_length, realtype *v_data);`

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

(This function does *not* allocate memory for `v_data` itself.)

F2003 Name This function is callable as `FN_VMake_Serial` when using the Fortran 2003 interface module.

N_VCloneVectorArray_Serial

Prototype `N_Vector *N_VCloneVectorArray_Serial(int count, N_Vector w);`

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

F2003 Name This function is callable as `FN_VCloneVectorArray_Serial` when using the Fortran 2003 interface module.

N_VCloneVectorArrayEmpty_Serial

Prototype `N_Vector *N_VCloneVectorArrayEmpty_Serial(int count, N_Vector w);`

Description This function creates (by cloning) an array of `count` serial vectors, each with an empty (NULL) data array.

F2003 Name This function is callable as `FN_VCloneVectorArrayEmpty_Serial` when using the Fortran 2003 interface module.

N_VDestroyVectorArray_Serial

Prototype `void N_VDestroyVectorArray_Serial(N_Vector *vs, int count);`

Description 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`.

F2003 Name This function is callable as `FN_VDestroyVectorArray_Serial` when using the Fortran 2003 interface module.

N_VPrint_Serial

Prototype `void N_VPrint_Serial(N_Vector v);`

Description This function prints the content of a serial vector to `stdout`.

F2003 Name This function is callable as `FN_VPrint_Serial` when using the Fortran 2003 interface module.

N_VPrintFile_Serial

Prototype `void N_VPrintFile_Serial(N_Vector v, FILE *outfile);`

Description This function prints the content of a serial vector to `outfile`.

F2003 Name This function is callable as `FN_VPrintFile_Serial` when using the Fortran 2003 interface module.

By default all fused and vector array operations are disabled in the `NVECTOR_SERIAL` module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_Serial`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone`. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VNew_Serial` will have the default settings for the `NVECTOR_SERIAL` module.

N_VEnableFusedOps_Serial

Prototype `int N_VEnableFusedOps_Serial(N_Vector v, booleantype tf);`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the serial vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are NULL.

F2003 Name This function is callable as `FN_VEnableFusedOps_Serial` when using the Fortran 2003 interface module.

N_VEnableLinearCombination_Serial

Prototype `int N_VEnableLinearCombination_Serial(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableLinearCombination_Serial` when using the Fortran 2003 interface module.

N_VEnableScaleAddMulti_Serial

Prototype `int N_VEnableScaleAddMulti_Serial(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableScaleAddMulti_Serial` when using the Fortran 2003 interface module.

N_VEnableDotProdMulti_Serial

Prototype `int N_VEnableDotProdMulti_Serial(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableDotProdMulti_Serial` when using the Fortran 2003 interface module.

N_VEnableLinearSumVectorArray_Serial

Prototype `int N_VEnableLinearSumVectorArray_Serial(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableLinearSumVectorArray_Serial` when using the Fortran 2003 interface module.

N_VEnableScaleVectorArray_Serial

Prototype `int N_VEnableScaleVectorArray_Serial(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableScaleVectorArray_Serial` when using the Fortran 2003 interface module.

N_VEnableConstVectorArray_Serial

Prototype `int N_VEnableConstVectorArray_Serial(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableConstVectorArray_Serial` when using the Fortran 2003 interface module.

`N_VEnableWrmsNormVectorArray_Serial`

Prototype `int N_VEnableWrmsNormVectorArray_Serial(N_Vector v, booleantype tf);`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the WRMS norm operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

F2003 Name This function is callable as `FN_VEnableWrmsNormVectorArray_Serial` when using the Fortran 2003 interface module.

`N_VEnableWrmsNormMaskVectorArray_Serial`

Prototype `int N_VEnableWrmsNormMaskVectorArray_Serial(N_Vector v, booleantype tf);`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the masked WRMS norm operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

F2003 Name This function is callable as `FN_VEnableWrmsNormMaskVectorArray_Serial` when using the Fortran 2003 interface module.

`N_VEnableScaleAddMultiVectorArray_Serial`

Prototype `int N_VEnableScaleAddMultiVectorArray_Serial(N_Vector v,
booleantype tf);`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector array to multiple vector arrays operation in the serial vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

`N_VEnableLinearCombinationVectorArray_Serial`

Prototype `int N_VEnableLinearCombinationVectorArray_Serial(N_Vector v,
booleantype tf);`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

Notes

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the component array via `v_data = NV_DATA_S(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_S(v,i)` within the loop.



- `N_VNewEmpty_Serial`, `N_VMake_Serial`, and `N_VCloneVectorArrayEmpty_Serial` set the field `own_data = SUNFALSE`. `N_VDestroy_Serial` and `N_VDestroyVectorArray_Serial` will not attempt to free the pointer `data` for any `N_Vector` with `own_data` set to `SUNFALSE`. In such a case, it is the user's responsibility to deallocate the `data` pointer.



- To maximize efficiency, vector operations in the `NVECTOR_SERIAL` implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

7.3.3 NVECTOR_SERIAL Fortran interfaces

The NVECTOR_SERIAL module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The `fnvector_serial_mod` FORTRAN module defines interfaces to all NVECTOR_SERIAL C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function `N_VNew_Serial` is interfaced as `FN_VNew_Serial`.

The FORTRAN 2003 NVECTOR_SERIAL interface module can be accessed with the `use` statement, i.e. `use fnvector_serial_mod`, and linking to the library `libsundials_fnvectorserial_mod.lib` in addition to the C library. For details on where the library and module file `fnvector_serial_mod.mod` are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators *without* separately linking to the `libsundials_fnvectorserial_mod` library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the NVECTOR_SERIAL module also includes a FORTRAN-callable function `FN_VINITs(code, NEQ, IER)`, to initialize this NVECTOR_SERIAL module. Here `code` is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); `NEQ` is the problem size (declared so as to match C type `long int`); and `IER` is an error return flag equal 0 for success and -1 for failure.

7.4 The NVECTOR_PARALLEL implementation

The NVECTOR_PARALLEL implementation of the NVECTOR module provided with SUNDIALS is based on MPI. It defines the `content` field of `N_Vector` to be a structure containing the global and local lengths of the vector, a pointer to the beginning of a contiguous local data array, an MPI communicator, and a boolean flag `own_data` indicating ownership of the data array `data`.

```
struct _N_VectorContent_Parallel {
    sunindextype local_length;
    sunindextype global_length;
    booleantype own_data;
    realtype *data;
    MPI_Comm comm;
};
```

The header file to include when using this module is `nvector_parallel.h`. The installed module library to link to is `libsundials_nvecparallel.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

7.4.1 NVECTOR_PARALLEL accessor macros

The following macros are provided to access the content of a NVECTOR_PARALLEL vector. The suffix `_P` in the names denotes the distributed memory parallel version.

- `NV_CONTENT_P`

This macro gives access to the contents of the parallel vector `N_Vector`.

The assignment `v_cont = NV_CONTENT_P(v)` sets `v_cont` to be a pointer to the `N_Vector` content structure of type `struct _N_VectorContent_Parallel`.

Implementation:

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

- NV_OWN_DATA_P, NV_DATA_P, NV_LOCLENGTH_P, NV_GLOBLENGTH_P

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

The assignment `v_data = NV_DATA_P(v)` sets `v_data` to be a pointer to the first component of the local data for the `N_Vector` `v`. The assignment `NV_DATA_P(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

The assignment `v_llen = NV_LOCLENGTH_P(v)` sets `v_llen` to be the length of the local part of `v`. The call `NV_LENGTH_P(v) = llen_v` sets the local length of `v` to be `llen_v`.

The assignment `v_glen = NV_GLOBLENGTH_P(v)` sets `v_glen` to be the global length of the vector `v`. The call `NV_GLOBLENGTH_P(v) = glen_v` sets the global length of `v` to be `glen_v`.

Implementation:

```
#define NV_OWN_DATA_P(v)    ( NV_CONTENT_P(v)->own_data )
#define NV_DATA_P(v)       ( NV_CONTENT_P(v)->data )
#define NV_LOCLENGTH_P(v)  ( NV_CONTENT_P(v)->local_length )
#define NV_GLOBLENGTH_P(v) ( NV_CONTENT_P(v)->global_length )
```

- NV_COMM_P

This macro provides access to the MPI communicator used by the `NVECTOR_PARALLEL` vectors.

Implementation:

```
#define NV_COMM_P(v) ( NV_CONTENT_P(v)->comm )
```

- NV_Ith_P

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

The assignment `r = NV_Ith_P(v,i)` sets `r` to be the value of the `i`-th component of the local part of `v`. The assignment `NV_Ith_P(v,i) = r` sets the value of the `i`-th component of the local part of `v` to be `r`.

Here `i` ranges from 0 to $n - 1$, where n is the local length.

Implementation:

```
#define NV_Ith_P(v,i) ( NV_DATA_P(v)[i] )
```

7.4.2 NVECTOR_PARALLEL functions

The `NVECTOR_PARALLEL` module defines parallel implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4. Their names are obtained from those in these tables by appending the suffix `_Parallel` (e.g. `N_VDestroy_Parallel`). The module `NVECTOR_PARALLEL` provides the following additional user-callable routines:

`N_VNew_Parallel`

Prototype `N_Vector N_VNew_Parallel(MPI_Comm comm, sunindextype local_length, sunindextype global_length);`

Description This function creates and allocates memory for a parallel vector.

F2003 Name This function is callable as `FN_VNew_Parallel` when using the Fortran 2003 interface module.

N_VNewEmpty_Parallel

Prototype `N_Vector N_VNewEmpty_Parallel(MPI_Comm comm, sunindextype local_length,
sunindextype global_length);`

Description This function creates a new parallel `N_Vector` with an empty (NULL) data array.

F2003 Name This function is callable as `FN_VNewEmpty_Parallel` when using the Fortran 2003 interface module.

N_VMake_Parallel

Prototype `N_Vector N_VMake_Parallel(MPI_Comm comm, sunindextype local_length,
sunindextype global_length, reatype *v_data);`

Description This function creates and allocates memory for a parallel vector with user-provided data array. This function does *not* allocate memory for `v_data` itself.

F2003 Name This function is callable as `FN_VMake_Parallel` when using the Fortran 2003 interface module.

N_VCloneVectorArray_Parallel

Prototype `N_Vector *N_VCloneVectorArray_Parallel(int count, N_Vector w);`

Description This function creates (by cloning) an array of `count` parallel vectors.

F2003 Name This function is callable as `FN_VCloneVectorArray_Parallel` when using the Fortran 2003 interface module.

N_VCloneVectorArrayEmpty_Parallel

Prototype `N_Vector *N_VCloneVectorArrayEmpty_Parallel(int count, N_Vector w);`

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

F2003 Name This function is callable as `FN_VCloneVectorArrayEmpty_Parallel` when using the Fortran 2003 interface module.

N_VDestroyVectorArray_Parallel

Prototype `void N_VDestroyVectorArray_Parallel(N_Vector *vs, int count);`

Description 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`.

F2003 Name This function is callable as `FN_VDestroyVectorArray_Parallel` when using the Fortran 2003 interface module.

N_VGetLocalLength_Parallel

Prototype `sunindextype N_VGetLocalLength_Parallel(N_Vector v);`

Description This function returns the local vector length.

F2003 Name This function is callable as `FN_VGetLocalLength_Parallel` when using the Fortran 2003 interface module.

N_VPrint_Parallel

Prototype `void N_VPrint_Parallel(N_Vector v);`

Description This function prints the local content of a parallel vector to `stdout`.

F2003 Name This function is callable as `FN_VPrint_Parallel` when using the Fortran 2003 interface module.

N_VPrintFile_Parallel

Prototype `void N_VPrintFile_Parallel(N_Vector v, FILE *outfile);`

Description This function prints the local content of a parallel vector to `outfile`.

F2003 Name This function is callable as `FN_VPrintFile_Parallel` when using the Fortran 2003 interface module.

By default all fused and vector array operations are disabled in the `NVECTOR_PARALLEL` module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_Parallel`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone` with that vector. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VNew_Parallel` will have the default settings for the `NVECTOR_PARALLEL` module.

N_VEnableFusedOps_Parallel

Prototype `int N_VEnableFusedOps_Parallel(N_Vector v, boolean_t tf);`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the parallel vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

F2003 Name This function is callable as `FN_VEnableFusedOps_Parallel` when using the Fortran 2003 interface module.

N_VEnableLinearCombination_Parallel

Prototype `int N_VEnableLinearCombination_Parallel(N_Vector v, boolean_t tf);`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the parallel vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

F2003 Name This function is callable as `FN_VEnableLinearCombination_Parallel` when using the Fortran 2003 interface module.

N_VEnableScaleAddMulti_Parallel

Prototype `int N_VEnableScaleAddMulti_Parallel(N_Vector v, boolean_t tf);`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector to multiple vectors fused operation in the parallel vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

F2003 Name This function is callable as `FN_VEnableScaleAddMulti_Parallel` when using the Fortran 2003 interface module.

N_VEnableDotProdMulti_Parallel

Prototype `int N_VEnableDotProdMulti_Parallel(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableDotProdMulti_Parallel` when using the Fortran 2003 interface module.

N_VEnableLinearSumVectorArray_Parallel

Prototype `int N_VEnableLinearSumVectorArray_Parallel(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableLinearSumVectorArray_Parallel` when using the Fortran 2003 interface module.

N_VEnableScaleVectorArray_Parallel

Prototype `int N_VEnableScaleVectorArray_Parallel(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableScaleVectorArray_Parallel` when using the Fortran 2003 interface module.

N_VEnableConstVectorArray_Parallel

Prototype `int N_VEnableConstVectorArray_Parallel(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableConstVectorArray_Parallel` when using the Fortran 2003 interface module.

N_VEnableWrmsNormVectorArray_Parallel

Prototype `int N_VEnableWrmsNormVectorArray_Parallel(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableWrmsNormVectorArray_Parallel` when using the Fortran 2003 interface module.

N_VEnableWrmsNormMaskVectorArray_Parallel

Prototype `int N_VEnableWrmsNormMaskVectorArray_Parallel(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableWrmsNormMaskVectorArray_Parallel` when using the Fortran 2003 interface module.

`N_VEnableScaleAddMultiVectorArray_Parallel`

Prototype `int N_VEnableScaleAddMultiVectorArray_Parallel(N_Vector v, booleantype tf);`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector array to multiple vector arrays operation in the parallel vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

`N_VEnableLinearCombinationVectorArray_Parallel`

Prototype `int N_VEnableLinearCombinationVectorArray_Parallel(N_Vector v, booleantype tf);`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

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.



7.4.3 NVECTOR_PARALLEL Fortran interfaces

For solvers that include a FORTRAN 77 interface module, the `NVECTOR_PARALLEL` module also includes a FORTRAN-callable function `FN_VINITP(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.



7.5 The NVECTOR_OPENMP implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, `SUNDIALS` provides an implementation of `NVECTOR` using OpenMP, called `NVECTOR_OPENMP`, and an implementation using Pthreads, called `NVECTOR_PTHREADS`. Testing has shown that vectors should be of length at least 100,000 before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The OpenMP NVECTOR implementation provided with SUNDIALS, NVECTOR_OPENMP, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag *own_data* which specifies the ownership of *data*, and the number of threads. Operations on the vector are threaded using OpenMP.

```
struct _N_VectorContent_OpenMP {
    sunindextype length;
    booleantype own_data;
    realtype *data;
    int num_threads;
};
```

The header file to include when using this module is `nvector_openmp.h`. The installed module library to link to is `libsundials_nvecopenmp.lib` where *.lib* is typically *.so* for shared libraries and *.a* for static libraries. The FORTRAN module file to use when using the FORTRAN 2003 interface to this module is `fnvector_openmp_mod.mod`.

7.5.1 NVECTOR_OPENMP accessor macros

The following macros are provided to access the content of an NVECTOR_OPENMP vector. The suffix *_OMP* in the names denotes the OpenMP version.

- `NV_CONTENT_OMP`

This routine gives access to the contents of the OpenMP vector `N_Vector`.

The assignment `v_cont = NV_CONTENT_OMP(v)` sets `v_cont` to be a pointer to the OpenMP `N_Vector` content structure.

Implementation:

```
#define NV_CONTENT_OMP(v) ( (N_VectorContent_OpenMP)(v->content) )
```

- `NV_OWN_DATA_OMP`, `NV_DATA_OMP`, `NV_LENGTH_OMP`, `NV_NUM_THREADS_OMP`

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

The assignment `v_data = NV_DATA_OMP(v)` sets `v_data` to be a pointer to the first component of the data for the `N_Vector` `v`. The assignment `NV_DATA_OMP(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

The assignment `v_len = NV_LENGTH_OMP(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_OMP(v) = len_v` sets the length of `v` to be `len_v`.

The assignment `v_num_threads = NV_NUM_THREADS_OMP(v)` sets `v_num_threads` to be the number of threads from `v`. On the other hand, the call `NV_NUM_THREADS_OMP(v) = num_threads_v` sets the number of threads for `v` to be `num_threads_v`.

Implementation:

```
#define NV_OWN_DATA_OMP(v) ( NV_CONTENT_OMP(v)->own_data )
```

```
#define NV_DATA_OMP(v) ( NV_CONTENT_OMP(v)->data )
```

```
#define NV_LENGTH_OMP(v) ( NV_CONTENT_OMP(v)->length )
```

```
#define NV_NUM_THREADS_OMP(v) ( NV_CONTENT_OMP(v)->num_threads )
```

- `NV_Ith_OMP`

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

The assignment `r = NV_Ith_OMP(v,i)` sets `r` to be the value of the *i*-th component of `v`. The assignment `NV_Ith_OMP(v,i) = r` sets the value of the *i*-th component of `v` to be `r`.

Here *i* ranges from 0 to *n* - 1 for a vector of length *n*.

Implementation:

```
#define NV_Ith_OMP(v,i) ( NV_DATA_OMP(v)[i] )
```

7.5.2 NVECTOR_OPENMP functions

The NVECTOR_OPENMP module defines OpenMP implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4. Their names are obtained from those in these tables by appending the suffix `_OpenMP` (e.g. `N_VDestroy_OpenMP`). All the standard vector operations listed in 7.1.1 with the suffix `_OpenMP` appended are callable via the FORTRAN 2003 interface by prepending an ‘F’ (e.g. `FN_VDestroy_OpenMP`).

The module NVECTOR_OPENMP provides the following additional user-callable routines:

N_VNew_OpenMP

Prototype `N_Vector N_VNew_OpenMP(sunindextype vec_length, int num_threads)`

Description This function creates and allocates memory for a OpenMP `N_Vector`. Arguments are the vector length and number of threads.

F2003 Name This function is callable as `FN_VNew_OpenMP` when using the Fortran 2003 interface module.

N_VNewEmpty_OpenMP

Prototype `N_Vector N_VNewEmpty_OpenMP(sunindextype vec_length, int num_threads)`

Description This function creates a new OpenMP `N_Vector` with an empty (NULL) data array.

F2003 Name This function is callable as `FN_VNewEmpty_OpenMP` when using the Fortran 2003 interface module.

N_VMake_OpenMP

Prototype `N_Vector N_VMake_OpenMP(sunindextype vec_length, realtype *v_data, int num_threads);`

Description This function creates and allocates memory for a OpenMP vector with user-provided data array. This function does *not* allocate memory for `v_data` itself.

F2003 Name This function is callable as `FN_VMake_OpenMP` when using the Fortran 2003 interface module.

N_VCloneVectorArray_OpenMP

Prototype `N_Vector *N_VCloneVectorArray_OpenMP(int count, N_Vector w)`

Description This function creates (by cloning) an array of `count` OpenMP vectors.

F2003 Name This function is callable as `FN_VCloneVectorArray_OpenMP` when using the Fortran 2003 interface module.

N_VCloneVectorArrayEmpty_OpenMP

Prototype `N_Vector *N_VCloneVectorArrayEmpty_OpenMP(int count, N_Vector w)`

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

F2003 Name This function is callable as `FN_VCloneVectorArrayEmpty_OpenMP` when using the Fortran 2003 interface module.

N_VDestroyVectorArray_OpenMP

Prototype `void N_VDestroyVectorArray_OpenMP(N_Vector *vs, int count)`

Description 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`.

F2003 Name This function is callable as `FN_VDestroyVectorArray_OpenMP` when using the Fortran 2003 interface module.

N_VPrint_OpenMP

Prototype `void N_VPrint_OpenMP(N_Vector v)`

Description This function prints the content of an OpenMP vector to `stdout`.

F2003 Name This function is callable as `FN_VPrint_OpenMP` when using the Fortran 2003 interface module.

N_VPrintFile_OpenMP

Prototype `void N_VPrintFile_OpenMP(N_Vector v, FILE *outfile)`

Description This function prints the content of an OpenMP vector to `outfile`.

F2003 Name This function is callable as `FN_VPrintFile_OpenMP` when using the Fortran 2003 interface module.

By default all fused and vector array operations are disabled in the `NVECTOR_OPENMP` module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_OpenMP`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone`. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VNew_OpenMP` will have the default settings for the `NVECTOR_OPENMP` module.

N_VEnableFusedOps_OpenMP

Prototype `int N_VEnableFusedOps_OpenMP(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

F2003 Name This function is callable as `FN_VEnableFusedOps_OpenMP` when using the Fortran 2003 interface module.

N_VEnableLinearCombination_OpenMP

Prototype `int N_VEnableLinearCombination_OpenMP(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

F2003 Name This function is callable as `FN_VEnableLinearCombination_OpenMP` when using the Fortran 2003 interface module.

N_VEnableScaleAddMulti_OpenMP

Prototype `int N_VEnableScaleAddMulti_OpenMP(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableScaleAddMulti_OpenMP when using the Fortran 2003 interface module.

N_VEnableDotProdMulti_OpenMP

Prototype `int N_VEnableDotProdMulti_OpenMP(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableDotProdMulti_OpenMP when using the Fortran 2003 interface module.

N_VEnableLinearSumVectorArray_OpenMP

Prototype `int N_VEnableLinearSumVectorArray_OpenMP(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableLinearSumVectorArray_OpenMP when using the Fortran 2003 interface module.

N_VEnableScaleVectorArray_OpenMP

Prototype `int N_VEnableScaleVectorArray_OpenMP(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableScaleVectorArray_OpenMP when using the Fortran 2003 interface module.

N_VEnableConstVectorArray_OpenMP

Prototype `int N_VEnableConstVectorArray_OpenMP(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableConstVectorArray_OpenMP when using the Fortran 2003 interface module.

N_VEnableWrmsNormVectorArray_OpenMP

Prototype `int N_VEnableWrmsNormVectorArray_OpenMP(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableWrmsNormVectorArray_OpenMP` when using the Fortran 2003 interface module.

`N_VEnableWrmsNormMaskVectorArray_OpenMP`

Prototype `int N_VEnableWrmsNormMaskVectorArray_OpenMP(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableWrmsNormMaskVectorArray_OpenMP` when using the Fortran 2003 interface module.

`N_VEnableScaleAddMultiVectorArray_OpenMP`

Prototype `int N_VEnableScaleAddMultiVectorArray_OpenMP(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

`N_VEnableLinearCombinationVectorArray_OpenMP`

Prototype `int N_VEnableLinearCombinationVectorArray_OpenMP(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

Notes

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the component array via `v_data = NV_DATA_OMP(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_OMP(v,i)` within the loop.
- `N_VNewEmpty_OpenMP`, `N_VMake_OpenMP`, and `N_VCloneVectorArrayEmpty_OpenMP` set the field `own_data = SUNFALSE`. `N_VDestroy_OpenMP` and `N_VDestroyVectorArray_OpenMP` will not attempt to free the pointer `data` for any `N_Vector` with `own_data` set to `SUNFALSE`. In such a case, it is the user's responsibility to deallocate the `data` pointer.
- To maximize efficiency, vector operations in the `NVECTOR_OPENMP` implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.



7.5.3 NVECTOR_OPENMP Fortran interfaces

The `NVECTOR_OPENMP` module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The `nvector_openmp_mod` FORTRAN module defines interfaces to most `NVECTOR_OPENMP` C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function `N_VNew_OpenMP` is interfaced as `FN_VNew_OpenMP`.

The FORTRAN 2003 NVECTOR_OPENMP interface module can be accessed with the `use` statement, i.e. `use fnvector_openmp_mod`, and linking to the library `libsundials_fnvectoropenmp_mod.lib` in addition to the C library. For details on where the library and module file `fnvector_openmp_mod.mod` are installed see Appendix A.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the NVECTOR_OPENMP module also includes a FORTRAN-callable function `FNINITOMP(code, NEQ, NUMTHREADS, IER)`, to initialize this module. Here `code` is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); `NEQ` is the problem size (declared so as to match C type `long int`); `NUMTHREADS` is the number of threads; and `IER` is an error return flag equal 0 for success and -1 for failure.

7.6 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.

7.6.1 NVECTOR_PTHREADS accessor macros

The following macros are provided to access the content of an NVECTOR_PTHREADS vector. The suffix `_PT` in the names denotes the Pthreads version.

- `NV_CONTENT_PT`

This routine gives access to the contents of the Pthreads vector `N_Vector`.

The assignment `v_cont = NV_CONTENT_PT(v)` sets `v_cont` to be a pointer to the Pthreads `N_Vector` content structure.

Implementation:

```
#define NV_CONTENT_PT(v) ( (N_VectorContent_Pthreads)(v->content) )
```

- `NV_OWN_DATA_PT`, `NV_DATA_PT`, `NV_LENGTH_PT`, `NV_NUM_THREADS_PT`

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

The assignment `v_data = NV_DATA_PT(v)` sets `v_data` to be a pointer to the first component of the data for the `N_Vector` `v`. The assignment `NV_DATA_PT(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

The assignment `v_len = NV_LENGTH_PT(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_PT(v) = len_v` sets the length of `v` to be `len_v`.

The assignment `v_num_threads = NV_NUM_THREADS_PT(v)` sets `v_num_threads` to be the number of threads from `v`. On the other hand, the call `NV_NUM_THREADS_PT(v) = num_threads_v` sets the number of threads for `v` to be `num_threads_v`.

Implementation:

```
#define NV_OWN_DATA_PT(v) ( NV_CONTENT_PT(v)->own_data )
#define NV_DATA_PT(v) ( NV_CONTENT_PT(v)->data )
#define NV_LENGTH_PT(v) ( NV_CONTENT_PT(v)->length )
#define NV_NUM_THREADS_PT(v) ( NV_CONTENT_PT(v)->num_threads )
```

- `NV_Ith_PT`

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

The assignment `r = NV_Ith_PT(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_PT(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

Here `i` ranges from 0 to `n - 1` for a vector of length `n`.

Implementation:

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

7.6.2 NVECTOR_PTHREADS functions

The `NVECTOR_PTHREADS` module defines Pthreads implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4. Their names are obtained from those in these tables by appending the suffix `_Pthreads` (e.g. `N_VDestroy_Pthreads`). All the standard vector operations listed in 7.1.1 are callable via the FORTRAN 2003 interface by prepending an ‘F’ (e.g. `FN_VDestroy_Pthreads`). The module `NVECTOR_PTHREADS` provides the following additional user-callable routines:

`N_VNew_Pthreads`

Prototype `N_Vector N_VNew_Pthreads(sunindextype vec_length, int num_threads)`

Description This function creates and allocates memory for a Pthreads `N_Vector`. Arguments are the vector length and number of threads.

F2003 Name This function is callable as `FN_VNew_Pthreads` when using the Fortran 2003 interface module.

`N_VNewEmpty_Pthreads`

Prototype `N_Vector N_VNewEmpty_Pthreads(sunindextype vec_length, int num_threads)`

Description This function creates a new Pthreads `N_Vector` with an empty (NULL) data array.

F2003 Name This function is callable as `FN_VNewEmpty_Pthreads` when using the Fortran 2003 interface module.

`N_VMake_Pthreads`

Prototype `N_Vector N_VMake_Pthreads(sunindextype vec_length, reatype *v_data, int num_threads);`

Description 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.

F2003 Name This function is callable as `FN_VMake_Pthreads` when using the Fortran 2003 interface module.

N_VCloneVectorArray_Pthreads

Prototype `N_Vector *N_VCloneVectorArray_Pthreads(int count, N_Vector w)`

Description This function creates (by cloning) an array of `count` Pthreads vectors.

F2003 Name This function is callable as `FN_VCloneVectorArray_Pthreads` when using the Fortran 2003 interface module.

N_VCloneVectorArrayEmpty_Pthreads

Prototype `N_Vector *N_VCloneVectorArrayEmpty_Pthreads(int count, N_Vector w)`

Description This function creates (by cloning) an array of `count` Pthreads vectors, each with an empty (NULL) data array.

F2003 Name This function is callable as `FN_VCloneVectorArrayEmpty_Pthreads` when using the Fortran 2003 interface module.

N_VDestroyVectorArray_Pthreads

Prototype `void N_VDestroyVectorArray_Pthreads(N_Vector *vs, int count)`

Description 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`.

F2003 Name This function is callable as `FN_VDestroyVectorArray_Pthreads` when using the Fortran 2003 interface module.

N_VPrint_Pthreads

Prototype `void N_VPrint_Pthreads(N_Vector v)`

Description This function prints the content of a Pthreads vector to `stdout`.

F2003 Name This function is callable as `FN_VPrint_Pthreads` when using the Fortran 2003 interface module.

N_VPrintFile_Pthreads

Prototype `void N_VPrintFile_Pthreads(N_Vector v, FILE *outfile)`

Description This function prints the content of a Pthreads vector to `outfile`.

F2003 Name This function is callable as `FN_VPrintFile_Pthreads` when using the Fortran 2003 interface module.

By default all fused and vector array operations are disabled in the `NVECTOR_PTHREADS` module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_Pthreads`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone`. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VNew_Pthreads` will have the default settings for the `NVECTOR_PTHREADS` module.

N_VEnableFusedOps_Pthreads

Prototype `int N_VEnableFusedOps_Pthreads(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableFusedOps_Pthreads` when using the Fortran 2003 interface module.

`N_VEnableLinearCombination_Pthreads`

Prototype `int N_VEnableLinearCombination_Pthreads(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableLinearCombination_Pthreads` when using the Fortran 2003 interface module.

`N_VEnableScaleAddMulti_Pthreads`

Prototype `int N_VEnableScaleAddMulti_Pthreads(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector to multiple vectors fused operation in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableScaleAddMulti_Pthreads` when using the Fortran 2003 interface module.

`N_VEnableDotProdMulti_Pthreads`

Prototype `int N_VEnableDotProdMulti_Pthreads(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the multiple dot products fused operation in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableDotProdMulti_Pthreads` when using the Fortran 2003 interface module.

`N_VEnableLinearSumVectorArray_Pthreads`

Prototype `int N_VEnableLinearSumVectorArray_Pthreads(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear sum operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableLinearSumVectorArray_Pthreads` when using the Fortran 2003 interface module.

`N_VEnableScaleVectorArray_Pthreads`

Prototype `int N_VEnableScaleVectorArray_Pthreads(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableScaleVectorArray_Pthreads` when using the Fortran 2003 interface module.

N_VEnableConstVectorArray_Pthreads

Prototype `int N_VEnableConstVectorArray_Pthreads(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableConstVectorArray_Pthreads` when using the Fortran 2003 interface module.

N_VEnableWrmsNormVectorArray_Pthreads

Prototype `int N_VEnableWrmsNormVectorArray_Pthreads(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableWrmsNormVectorArray_Pthreads` when using the Fortran 2003 interface module.

N_VEnableWrmsNormMaskVectorArray_Pthreads

Prototype `int N_VEnableWrmsNormMaskVectorArray_Pthreads(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableWrmsNormMaskVectorArray_Pthreads` when using the Fortran 2003 interface module.

N_VEnableScaleAddMultiVectorArray_Pthreads

Prototype `int N_VEnableScaleAddMultiVectorArray_Pthreads(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearCombinationVectorArray_Pthreads

Prototype `int N_VEnableLinearCombinationVectorArray_Pthreads(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

Notes

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the component array via `v_data = NV_DATA_PT(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_PT(v, i)` within the loop.
- `N_VNewEmpty_Pthreads`, `N_VMake_Pthreads`, and `N_VCloneVectorArrayEmpty_Pthreads` set the field `own_data = SUNFALSE`. `N_VDestroy_Pthreads` and `N_VDestroyVectorArray_Pthreads` will not attempt to free the pointer `data` for any `N_Vector` with `own_data` set to `SUNFALSE`. In such a case, it is the user's responsibility to deallocate the `data` pointer.





- To maximize efficiency, vector operations in the NVECTOR_PTHREADS implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

7.6.3 NVECTOR_PTHREADS Fortran interfaces

The NVECTOR_PTHREADS module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The `nvector_pthreads_mod` FORTRAN module defines interfaces to most NVECTOR_PTHREADS C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function `N_VNew_Pthreads` is interfaced as `FN_VNew_Pthreads`.

The FORTRAN 2003 NVECTOR_PTHREADS interface module can be accessed with the `use` statement, i.e. `use fnvector_pthreads_mod`, and linking to the library `libsundials_fnvectorpthreads_mod.lib` in addition to the C library. For details on where the library and module file `fnvector_pthreads_mod.mod` are installed see [Appendix A](#).

FORTRAN 77 interface functions

For solvers that include a FORTRAN interface module, the NVECTOR_PTHREADS module also includes a FORTRAN-callable function `FNVINITPTS(code, NEQ, NUMTHREADS, IER)`, to initialize this module. Here `code` is an input solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, 4 for `ARKODE`); `NEQ` is the problem size (declared so as to match C type `long int`); `NUMTHREADS` is the number of threads; and `IER` is an error return flag equal 0 for success and -1 for failure.

7.7 The NVECTOR_PARHYP implementation

The NVECTOR_PARHYP implementation of the NVECTOR module provided with SUNDIALS is a wrapper around `hypre`'s `ParVector` class. Most of the vector kernels simply call `hypre` vector operations. The implementation defines the `content` field of `N_Vector` to be a structure containing the global and local lengths of the vector, a pointer to an object of type `HYPRE_ParVector`, an MPI communicator, and a boolean flag `own_parvector` indicating ownership of the `hypre` parallel vector object `x`.

```
struct _N_VectorContent_ParHyp {
    sunindextype local_length;
    sunindextype global_length;
    booleantype own_parvector;
    MPI_Comm comm;
    HYPRE_ParVector x;
};
```

The header file to include when using this module is `nvector_parhyp.h`. The installed module library to link to is `libsundials_nvecparhyp.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

Unlike native SUNDIALS vector types, NVECTOR_PARHYP does not provide macros to access its member variables. Note that NVECTOR_PARHYP requires SUNDIALS to be built with MPI support.

7.7.1 NVECTOR_PARHYP functions

The NVECTOR_PARHYP module defines implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4, except for `N_VSetArrayPointer` and `N_VGetArrayPointer`, because accessing raw vector data is handled by low-level *hypre* functions. As such, this vector is not available for use with SUNDIALS Fortran interfaces. When access to raw vector data is needed, one should extract the *hypre* vector first, and then use *hypre* methods to access the data. Usage examples of NVECTOR_PARHYP are provided in the `cvAdvDiff_non_ph.c` example program for CVODE [28] and the `ark_diurnal_kry_ph.c` example program for ARKODE [36].

The names of parhyp methods are obtained from those in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4 by appending the suffix `_ParHyp` (e.g. `N_VDestroy_ParHyp`). The module NVECTOR_PARHYP provides the following additional user-callable routines:

`N_VNewEmpty_ParHyp`

Prototype `N_Vector N_VNewEmpty_ParHyp(MPI_Comm comm, sunindextype local_length, sunindextype global_length)`

Description This function creates a new parhyp `N_Vector` with the pointer to the *hypre* vector set to NULL.

`N_VMake_ParHyp`

Prototype `N_Vector N_VMake_ParHyp(HYPRE_ParVector x)`

Description This function creates an `N_Vector` wrapper around an existing *hypre* parallel vector. It does *not* allocate memory for `x` itself.

`N_VGetVector_ParHyp`

Prototype `HYPRE_ParVector N_VGetVector_ParHyp(N_Vector v)`

Description This function returns the underlying *hypre* vector.

`N_VCloneVectorArray_ParHyp`

Prototype `N_Vector *N_VCloneVectorArray_ParHyp(int count, N_Vector w)`

Description This function creates (by cloning) an array of `count` parallel vectors.

`N_VCloneVectorArrayEmpty_ParHyp`

Prototype `N_Vector *N_VCloneVectorArrayEmpty_ParHyp(int count, N_Vector w)`

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

`N_VDestroyVectorArray_ParHyp`

Prototype `void N_VDestroyVectorArray_ParHyp(N_Vector *vs, int count)`

Description 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`.

`N_VPrint_ParHyp`

Prototype `void N_VPrint_ParHyp(N_Vector v)`

Description This function prints the local content of a parhyp vector to `stdout`.

N_VPrintFile_ParHyp

Prototype `void N_VPrintFile_ParHyp(N_Vector v, FILE *outfile)`

Description This function prints the local content of a parhyp vector to `outfile`.

By default all fused and vector array operations are disabled in the NVECTOR_PARHYP module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VMake_ParHyp`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone`. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VMake_ParHyp` will have the default settings for the NVECTOR_PARHYP module.

N_VEnableFusedOps_ParHyp

Prototype `int N_VEnableFusedOps_ParHyp(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the parhyp vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

N_VEnableLinearCombination_ParHyp

Prototype `int N_VEnableLinearCombination_ParHyp(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the parhyp vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

N_VEnableScaleAddMulti_ParHyp

Prototype `int N_VEnableScaleAddMulti_ParHyp(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector to multiple vectors fused operation in the parhyp vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

N_VEnableDotProdMulti_ParHyp

Prototype `int N_VEnableDotProdMulti_ParHyp(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the multiple dot products fused operation in the parhyp vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

N_VEnableLinearSumVectorArray_ParHyp

Prototype `int N_VEnableLinearSumVectorArray_ParHyp(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear sum operation for vector arrays in the parhyp vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

N_VEnableScaleVectorArray_ParHyp

Prototype `int N_VEnableScaleVectorArray_ParHyp(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableConstVectorArray_ParHyp

Prototype int N_VEnableConstVectorArray_ParHyp(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableWrmsNormVectorArray_ParHyp

Prototype int N_VEnableWrmsNormVectorArray_ParHyp(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableWrmsNormMaskVectorArray_ParHyp

Prototype int N_VEnableWrmsNormMaskVectorArray_ParHyp(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableScaleAddMultiVectorArray_ParHyp

Prototype int N_VEnableScaleAddMultiVectorArray_ParHyp(N_Vector v,
booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearCombinationVectorArray_ParHyp

Prototype int N_VEnableLinearCombinationVectorArray_ParHyp(N_Vector v,
booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

Notes

- When there is a need to access components of an N_Vector_ParHyp, v, it is recommended to extract the *hypre* vector via `x_vec = N_VGetVector_ParHyp(v)` and then access components using appropriate *hypre* functions.



- N_VNewEmpty_ParHyp, N_VMake_ParHyp, and N_VCloneVectorArrayEmpty_ParHyp set the field *own_parvector* to SUNFALSE. N_VDestroy_ParHyp and N_VDestroyVectorArray_ParHyp will not attempt to delete an underlying *hypre* vector for any N_Vector with *own_parvector* set to SUNFALSE. In such a case, it is the user's responsibility to delete the underlying vector.



- To maximize efficiency, vector operations in the NVECTOR_PARHYP implementation that have more than one `N_Vector` argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

7.8 The NVECTOR_PETSC implementation

The NVECTOR_PETSC module is an NVECTOR wrapper around the PETSc vector. It defines the *content* field of a `N_Vector` to be a structure containing the global and local lengths of the vector, a pointer to the PETSc vector, an MPI communicator, and a boolean flag *own_data* indicating ownership of the wrapped PETSc vector.

```
struct _N_VectorContent_Petsc {
    sunindextype local_length;
    sunindextype global_length;
    booleantype own_data;
    Vec *pvec;
    MPI_Comm comm;
};
```

The header file to include when using this module is `nvector_petsc.h`. The installed module library to link to is `libsundials_nvecpetsc.lib` where *.lib* is typically *.so* for shared libraries and *.a* for static libraries.

Unlike native SUNDIALS vector types, NVECTOR_PETSC does not provide macros to access its member variables. Note that NVECTOR_PETSC requires SUNDIALS to be built with MPI support.

7.8.1 NVECTOR_PETSC functions

The NVECTOR_PETSC module defines implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4, 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 [27].

The names of vector operations are obtained from those in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4 by appending the suffix `_Petsc` (e.g. `N_VDestroy_Petsc`). The module NVECTOR_PETSC provides the following additional user-callable routines:

`N_VNewEmpty_Petsc`

Prototype `N_Vector N_VNewEmpty_Petsc(MPI_Comm comm, sunindextype local_length, sunindextype global_length)`

Description This function creates a new NVECTOR wrapper with the pointer to the wrapped PETSc vector set to (NULL). It is used by the `N_VMake_Petsc` and `N_VClone_Petsc` implementations.

`N_VMake_Petsc`

Prototype `N_Vector N_VMake_Petsc(Vec *pvec)`

Description This function creates and allocates memory for an NVECTOR_PETSC wrapper around a user-provided PETSc vector. It does *not* allocate memory for the vector `pvec` itself.

`N_VGetVector_Petsc`

Prototype `Vec *N_VGetVector_Petsc(N_Vector v)`

Description This function returns a pointer to the underlying PETSc vector.

N_VCloneVectorArray_Petsc

Prototype `N_Vector *N_VCloneVectorArray_Petsc(int count, N_Vector w)`

Description This function creates (by cloning) an array of `count` `NVECTOR_PETSC` vectors.

N_VCloneVectorArrayEmpty_Petsc

Prototype `N_Vector *N_VCloneVectorArrayEmpty_Petsc(int count, N_Vector w)`

Description This function creates (by cloning) an array of `count` `NVECTOR_PETSC` vectors, each with pointers to PETSc vectors set to (NULL).

N_VDestroyVectorArray_Petsc

Prototype `void N_VDestroyVectorArray_Petsc(N_Vector *vs, int count)`

Description 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`.

N_VPrint_Petsc

Prototype `void N_VPrint_Petsc(N_Vector v)`

Description This function prints the global content of a wrapped PETSc vector to `stdout`.

N_VPrintFile_Petsc

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

Description This function prints the global content of a wrapped PETSc vector to `fname`.

By default all fused and vector array operations are disabled in the `NVECTOR_PETSC` module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VMake_Petsc`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone`. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VMake_Petsc` will have the default settings for the `NVECTOR_PETSC` module.

N_VEnableFusedOps_Petsc

Prototype `int N_VEnableFusedOps_Petsc(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the PETSc vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are NULL.

N_VEnableLinearCombination_Petsc

Prototype `int N_VEnableLinearCombination_Petsc(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the PETSc vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are NULL.

N_VEnableScaleAddMulti_Petsc

Prototype `int N_VEnableScaleAddMulti_Petsc(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableDotProdMulti_Petsc

Prototype `int N_VEnableDotProdMulti_Petsc(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearSumVectorArray_Petsc

Prototype `int N_VEnableLinearSumVectorArray_Petsc(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableScaleVectorArray_Petsc

Prototype `int N_VEnableScaleVectorArray_Petsc(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableConstVectorArray_Petsc

Prototype `int N_VEnableConstVectorArray_Petsc(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableWrmsNormVectorArray_Petsc

Prototype `int N_VEnableWrmsNormVectorArray_Petsc(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableWrmsNormMaskVectorArray_Petsc

Prototype `int N_VEnableWrmsNormMaskVectorArray_Petsc(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableScaleAddMultiVectorArray_Petsc

Prototype `int N_VEnableScaleAddMultiVectorArray_Petsc(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the PETSC vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearCombinationVectorArray_Petsc

Prototype `int N_VEnableLinearCombinationVectorArray_Petsc(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the PETSC vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

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 `NVECTOR_PETSC` implementation that have more than one `N_Vector` argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.



7.9 The NVECTOR_CUDA implementation

The `NVECTOR_CUDA` module is an `NVECTOR` implementation in the CUDA language. The module allows for SUNDIALS vector kernels to run on NVIDIA 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 vector content layout is as follows:

```
struct _N_VectorContent_Cuda
{
    sunindextype    length;
    booleantype    own_exec;
    booleantype    own_helper;
    SUNMemory      host_data;
    SUNMemory      device_data;
    SUNCudaExecPolicy* stream_exec_policy;
    SUNCudaExecPolicy* reduce_exec_policy;
    SUNMemoryHelper mem_helper;
    void*          priv; /* 'private' data */
};

typedef struct _N_VectorContent_Cuda *N_VectorContent_Cuda;
```

The content members are the vector length (size), ownership flags for the `*_exec_policy` fields and the `mem_helper` field, `SUNMemory` objects for the vector data on the host and the device, pointers to

SUNCudaExecPolicy implementations that control how the CUDA kernels are launched for streaming and reduction vector kernels, a SUNMemoryHelper object, and a private data structure which holds additional members that should not be accessed directly.

When instantiated with N_VNew_Cuda, the underlying data will be allocated memory on both the host and the device. Alternatively, a user can provide host and device data arrays by using the N_VMake_Cuda constructor. To use CUDA managed memory, the constructors N_VNewManaged_Cuda and

N_VMakeManaged_Cuda are provided. Details on each of these constructors are provided below.

To use the NVECTOR_CUDA module, the header file to include is `nvector_cuda.h`, and the library to link to is `libsundials_nveccuda.lib`. The extension `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

7.9.1 NVECTOR_CUDA functions

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

N_VGetHostArrayPointer_Cuda

Prototype `realtype *N_VGetHostArrayPointer_Cuda(N_Vector v)`

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

N_VGetDeviceArrayPointer_Cuda

Prototype `realtype *N_VGetDeviceArrayPointer_Cuda(N_Vector v)`

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

N_VSetHostArrayPointer_Cuda

Prototype `realtype *N_VSetHostArrayPointer_Cuda(N_Vector v)`

Description This function sets the pointer to the vector data on the host. The existing pointer *will not* be freed first.

N_VSetDeviceArrayPointer_Cuda

Prototype `realtype *N_VSetDeviceArrayPointer_Cuda(N_Vector v)`

Description This function sets pointer to the vector data on the device. The existing pointer *will not* be freed first.

N_VIsManagedMemory_Cuda

Prototype `boolean_t *N_VIsManagedMemory_Cuda(N_Vector v)`

Description This function returns a boolean flag indicating if the vector data is allocated in managed memory or not.

The NVECTOR_CUDA module defines implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3 and 7.1.4, except for `N_VSetArrayPointer` and `N_VGetArrayPointer` unless managed memory is used. As such, this vector can only be used with the SUNDIALS Fortran interfaces, and the SUNDIALS direct solvers and preconditioners when using managed memory. The NVECTOR_CUDA module provides separate functions to access data on the host and on the device for the unmanaged memory use case. 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 [28].

The names of vector operations are obtained from those in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4 by appending the suffix `_Cuda` (e.g. `N_VDestroy_Cuda`). The module NVECTOR_CUDA provides the following functions:

N_VNew_Cuda

Prototype `N_Vector N_VNew_Cuda(sunindextype length)`

Description This function creates and allocates memory for a CUDA `N_Vector`. The vector data array is allocated on both the host and device.

N_VNewManaged_Cuda

Prototype `N_Vector N_VNewManaged_Cuda(sunindextype length)`

Description This function creates and allocates memory for a CUDA `N_Vector`. The vector data array is allocated in managed memory.

N_VNewWithMemHelp_Cuda

Prototype `N_Vector N_VNewWithMemHelp_Cuda(sunindextype length, booleantype use_managed_mem, SUNMemoryHelper helper);`

Description This function creates an `NVECTOR_CUDA` which will use the `SUNMemoryHelper` object to allocate memory. If `use_managed_memory` is 0, then unmanaged memory is used, otherwise managed memory is used.

N_VNewEmpty_Cuda

Prototype `N_Vector N_VNewEmpty_Cuda()`

Description This function creates a new `NVECTOR` wrapper with the pointer to the wrapped CUDA vector set to `NULL`. It is used by the `N_VNew_Cuda`, `N_VMake_Cuda`, and `N_VClone_Cuda` implementations.

N_VMake_Cuda

Prototype `N_Vector N_VMake_Cuda(sunindextype length, reatype *h_data, reatype *dev_data)`

Description This function creates an `NVECTOR_CUDA` with user-supplied vector data arrays `h_vdata` and `d_vdata`. This function does not allocate memory for data itself.

N_VMakeManaged_Cuda

Prototype `N_Vector N_VMakeManaged_Cuda(sunindextype length, reatype *vdata)`

Description This function creates an `NVECTOR_CUDA` with a user-supplied managed memory data array. This function does not allocate memory for data itself.

N_VMakeWithManagedAllocator_Cuda

Prototype `N_Vector N_VMakeWithManagedAllocator_Cuda(sunindextype length, void* (*allocfn)(size_t size), void (*freefn)(void* ptr));`

Description This function creates an `NVECTOR_CUDA` with a user-supplied memory allocator. It requires the user to provide a corresponding free function as well. The memory allocated by the allocator function must behave like CUDA managed memory.

This function is deprecated and will be removed in the next major release. Use `N_VNewWithMemHelp_Cuda` instead.

The module `NVECTOR_CUDA` also provides the following user-callable routines:



N_VSetKernelExecPolicy_Cuda

Prototype `void N_VSetKernelExecPolicy_Cuda(N_Vector v, SUNCudaExecPolicy* stream_exec_policy, SUNCudaExecPolicy* reduce_exec_policy);`

Description This function sets the execution policies which control the kernel parameters utilized when launching the streaming and reduction CUDA kernels. By default the vector is setup to use the `SUNCudaThreadDirectExecPolicy` and `SUNCudaBlockReduceExecPolicy`. Any custom execution policy for reductions must ensure that the grid dimensions (number of thread blocks) is a multiple of the CUDA warp size (32). See section 7.9.2 below for more information about the `SUNCudaExecPolicy` class.

*Note: All vectors used in a single instance of a SUNDIALS solver must use the same execution policy. It is **strongly recommended** that this function is called immediately after constructing the vector, and any subsequent vector be created by cloning to ensure consistent execution policies across vectors.*

N_VSetCudaStream_Cuda

Prototype `void N_VSetCudaStream_Cuda(N_Vector v, cudaStream_t *stream)`

Description This function sets the CUDA stream that all vector kernels will be launched on. By default an NVECTOR_CUDA uses the default CUDA stream.

*Note: All vectors used in a single instance of a SUNDIALS solver must use the same CUDA stream. It is **strongly recommended** that this function is called immediately after constructing the vector, and any subsequent vector be created by cloning to ensure consistent execution policies across vectors.*

This function will be removed in the next major release, user should utilize the `N_VSetKernelExecPolicy_Cuda` function instead. 

N_VCopyToDevice_Cuda

Prototype `void N_VCopyToDevice_Cuda(N_Vector v)`

Description This function copies host vector data to the device.

N_VCopyFromDevice_Cuda

Prototype `void N_VCopyFromDevice_Cuda(N_Vector v)`

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

N_VPrint_Cuda

Prototype `void N_VPrint_Cuda(N_Vector v)`

Description This function prints the content of a CUDA vector to `stdout`.

N_VPrintFile_Cuda

Prototype `void N_VPrintFile_Cuda(N_Vector v, FILE *outfile)`

Description This function prints the content of a CUDA vector to `outfile`.

By default all fused and vector array operations are disabled in the NVECTOR_CUDA module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_Cuda`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone`. This guarantees

the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VNew_Cuda` will have the default settings for the `NVECTOR_CUDA` module.

`N_VEnableFusedOps_Cuda`

Prototype `int N_VEnableFusedOps_Cuda(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the `CUDA` vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

`N_VEnableLinearCombination_Cuda`

Prototype `int N_VEnableLinearCombination_Cuda(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the `CUDA` vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

`N_VEnableScaleAddMulti_Cuda`

Prototype `int N_VEnableScaleAddMulti_Cuda(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector to multiple vectors fused operation in the `CUDA` vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

`N_VEnableDotProdMulti_Cuda`

Prototype `int N_VEnableDotProdMulti_Cuda(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the multiple dot products fused operation in the `CUDA` vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

`N_VEnableLinearSumVectorArray_Cuda`

Prototype `int N_VEnableLinearSumVectorArray_Cuda(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear sum operation for vector arrays in the `CUDA` vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

`N_VEnableScaleVectorArray_Cuda`

Prototype `int N_VEnableScaleVectorArray_Cuda(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale operation for vector arrays in the `CUDA` vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

`N_VEnableConstVectorArray_Cuda`

Prototype `int N_VEnableConstVectorArray_Cuda(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the const operation for vector arrays in the `CUDA` vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

N.EnableWrmsNormVectorArray_Cuda

Prototype `int N.EnableWrmsNormVectorArray_Cuda(N.Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N.EnableWrmsNormMaskVectorArray_Cuda

Prototype `int N.EnableWrmsNormMaskVectorArray_Cuda(N.Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N.EnableScaleAddMultiVectorArray_Cuda

Prototype `int N.EnableScaleAddMultiVectorArray_Cuda(N.Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N.EnableLinearCombinationVectorArray_Cuda

Prototype `int N.EnableLinearCombinationVectorArray_Cuda(N.Vector v,
booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

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`. However, when using managed memory, the function `N_VGetArrayPointer` may also be used.
- Performance is better if the `SUNMemoryHelper` provided supports `SUNMEMTYPE_PINNED`; the default `SUNMemoryHelper` does not provide this support. In the case that it does, then the buffers used for reductions will be allocated as pinned memory.
- 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.

**7.9.2 The SUNCudaExecPolicy Class**

In order to provide maximum flexibility to users, the CUDA kernel execution parameters used by kernels within SUNDIALS are defined by objects of the `sundials::CudaExecPolicy` abstract class type (this class can be accessed in the global namespace as `SUNCudaExecPolicy`). Thus, users may provide custom execution policies that fit the needs of their problem. The `sundials::CudaExecPolicy` is defined in the header file `sundials_cuda_policies.hpp`, and is as follows:

```
class CudaExecPolicy
{
public:
```

```

virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const = 0;
virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const = 0;
virtual cudaStream_t stream() const = 0;
virtual CudaExecPolicy* clone() const = 0;
virtual ~CudaExecPolicy() {}
};

```

To define a custom execution policy, a user simply needs to create a class that inherits from the abstract class and implements the methods. The SUNDIALS provided `sundials::CudaThreadDirectExecPolicy` (aka in the global namespace as `SUNCudaThreadDirectExecPolicy`) class is a good example of a what a custom execution policy may look like:

```

class CudaThreadDirectExecPolicy : public CudaExecPolicy
{
public:
    CudaThreadDirectExecPolicy(const size_t blockDim, const cudaStream_t stream = 0)
        : blockDim_(blockDim), stream_(stream)
    {}

    CudaThreadDirectExecPolicy(const CudaThreadDirectExecPolicy& ex)
        : blockDim_(ex.blockDim_), stream_(ex.stream_)
    {}

    virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const
    {
        return (numWorkUnits + blockSize() - 1) / blockSize();
    }

    virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const
    {
        return blockDim_;
    }

    virtual cudaStream_t stream() const
    {
        return stream_;
    }

    virtual CudaExecPolicy* clone() const
    {
        return static_cast<CudaExecPolicy*>(new CudaThreadDirectExecPolicy(*this));
    }

private:
    const cudaStream_t stream_;
    const size_t blockDim_;
};

```

In total, SUNDIALS provides 3 execution policies:

1. `SUNCudaThreadDirectExecPolicy(const size_t blockDim, const cudaStream_t stream = 0)` maps each CUDA thread to a work unit. The number of threads per block (`blockDim`) can be set to anything. The grid size will be calculated so that there are enough threads for one thread per element. If a CUDA stream is provided, it will be used to execute the kernel.

2. `SUNCudaGridStrideExecPolicy(const size_t blockDim, const size_t gridDim, const cudaStream_t stream = 0)` is for kernels that use grid stride loops. The number of threads per block (`blockDim`) can be set to anything. The number of blocks (`gridDim`) can be set to anything. If a CUDA stream is provided, it will be used to execute the kernel.
3. `SUNCudaBlockReduceExecPolicy(const size_t blockDim, const size_t gridDim, const cudaStream_t stream = 0)` is for kernels performing a reduction across individual thread blocks. The number of threads per block (`blockDim`) can be set to any valid multiple of the CUDA warp size. The grid size (`gridDim`) can be set to any value greater than 0. If it is set to 0, then the grid size will be chosen so that there is enough threads for one thread per work unit. If a CUDA stream is provided, it will be used to execute the kernel.

For example, a policy that uses 128 threads per block and a user provided stream can be created like so:

```
cudaStream_t stream;
cudaStreamCreate(&stream);
SUNCudaThreadDirectExecPolicy thread_direct(128, stream);
```

These default policy objects can be reused for multiple SUNDIALS data structures since they do not hold any modifiable state information.

7.10 The NVECTOR_HIP implementation

The `NVECTOR_HIP` module is an `NVECTOR` implementation using the AMD ROCm HIP library. The module allows for SUNDIALS vector kernels to run on AMD or NVIDIA GPU devices. It is intended for users who are already familiar with HIP and GPU programming. Building this vector module requires the HIP-clang compiler. The vector content layout is as follows:

```
struct _N_VectorContent_Hip
{
    sunindextype    length;
    booleantype    own_exec;
    booleantype    own_helper;
    SUNMemory      host_data;
    SUNMemory      device_data;
    SUNHipExecPolicy* stream_exec_policy;
    SUNHipExecPolicy* reduce_exec_policy;
    SUNMemoryHelper mem_helper;
    void*          priv; /* 'private' data */
};

typedef struct _N_VectorContent_Hip *N_VectorContent_Hip;
```

The content members are the vector length (size), a boolean flag that signals if the vector owns the data (i.e. it is in charge of freeing the data), pointers to vector data on the host and the device, pointers to `SUNHipExecPolicy` implementations that control how the HIP kernels are launched for streaming and reduction vector kernels, and a private data structure which holds additional members that should not be accessed directly.

When instantiated with `N_VNew_Hip`, the underlying data will be allocated memory on both the host and the device. Alternatively, a user can provide host and device data arrays by using the `N_VMake_Hip` constructor. To use HIP managed memory, the constructors `N_VNewManaged_Hip` and `N_VMakeManaged_Hip` are provided. Details on each of these constructors are provided below.

To use the `NVECTOR_HIP` module, the header file to include is `nvector_hip.h`, and the library to link to is `libsundials_nvechip.lib`. The extension `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

7.10.1 NVECTOR_HIP functions

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

N_VGetHostArrayPointer_Hip

Prototype `realtype *N_VGetHostArrayPointer_Hip(N_Vector v)`

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

N_VGetDeviceArrayPointer_Hip

Prototype `realtype *N_VGetDeviceArrayPointer_Hip(N_Vector v)`

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

N_VIsManagedMemory_Hip

Prototype `boolean_t *N_VIsManagedMemory_Hip(N_Vector v)`

Description This function returns a boolean flag indicating if the vector data is allocated in managed memory or not.

The NVECTOR_HIP module defines implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3 and 7.1.4, except for `N_VSetArrayPointer`. The names of vector operations are obtained from those in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4 by appending the suffix `_Hip` (e.g. `N_VDestroy_Hip`). The module NVECTOR_HIP provides the following functions:

N_VNew_Hip

Prototype `N_Vector N_VNew_Hip(sunindextype length)`

Description This function creates an empty HIP `N_Vector` with the data pointers set to `NULL`.

N_VNewManaged_Hip

Prototype `N_Vector N_VNewManaged_Hip(sunindextype length)`

Description This function creates and allocates memory for a HIP `N_Vector`. The vector data array is allocated in managed memory.

N_VNewEmpty_Hip

Prototype `N_Vector N_VNewEmpty_Hip()`

Description This function creates a new NVECTOR wrapper with the pointer to the wrapped HIP vector set to `NULL`. It is used by the `N_VNew_Hip`, `N_VMake_Hip`, and `N_VClone_Hip` implementations.

N_VMake_Hip

Prototype `N_Vector N_VMake_Hip(sunindextype length, realtype *h_data, realtype *dev_data)`

Description This function creates an NVECTOR_HIP with user-supplied vector data arrays `h_vdata` and `d_vdata`. This function does not allocate memory for data itself.

N_VMakeManaged_Hip

Prototype `N_Vector N_VMakeManaged_Hip(sunindextype length, reatype *vdata)`

Description This function creates an NVECTOR_HIP with a user-supplied managed memory data array. This function does not allocate memory for data itself.

The module NVECTOR_HIP also provides the following user-callable routines:

N_VSetKernelExecPolicy_Hip

Prototype `void N_VSetKernelExecPolicy_Hip(N_Vector v,
SUNHipExecPolicy* stream_exec_policy,
SUNHipExecPolicy* reduce_exec_policy);`

Description This function sets the execution policies which control the kernel parameters utilized when launching the streaming and reduction HIP kernels. By default the vector is setup to use the `SUNHipThreadDirectExecPolicy` and `SUNHipBlockReduceExecPolicy`. Any custom execution policy for reductions must ensure that the grid dimensions (number of thread blocks) is a multiple of the HIP warp size (64 when targeting AMD GPUs and 32 when targeting NVIDIA GPUs). See section 7.10.2 below for more information about the `SUNHipExecPolicy` class.

*Note: All vectors used in a single instance of a SUNDIALS solver must use the same execution policy. It is **strongly recommended** that this function is called immediately after constructing the vector, and any subsequent vector be created by cloning to ensure consistent execution policies across vectors.*

N_VCopyToDevice_Hip

Prototype `void N_VCopyToDevice_Hip(N_Vector v)`

Description This function copies host vector data to the device.

N_VCopyFromDevice_Hip

Prototype `void N_VCopyFromDevice_Hip(N_Vector v)`

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

N_VPrint_Hip

Prototype `void N_VPrint_Hip(N_Vector v)`

Description This function prints the content of a HIP vector to `stdout`.

N_VPrintFile_Hip

Prototype `void N_VPrintFile_Hip(N_Vector v, FILE *outfile)`

Description This function prints the content of a HIP vector to `outfile`.

By default all fused and vector array operations are disabled in the NVECTOR_HIP module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_Hip`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone`. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VNew_Hip` will have the default settings for the NVECTOR_HIP module.

N_VEnableFusedOps_Hip

Prototype `int N_VEnableFusedOps_Hip(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the HIP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearCombination_Hip

Prototype `int N_VEnableLinearCombination_Hip(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the HIP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableScaleAddMulti_Hip

Prototype `int N_VEnableScaleAddMulti_Hip(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the HIP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableDotProdMulti_Hip

Prototype `int N_VEnableDotProdMulti_Hip(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the HIP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearSumVectorArray_Hip

Prototype `int N_VEnableLinearSumVectorArray_Hip(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the HIP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableScaleVectorArray_Hip

Prototype `int N_VEnableScaleVectorArray_Hip(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the HIP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableConstVectorArray_Hip

Prototype `int N_VEnableConstVectorArray_Hip(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the HIP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableWrmsNormVectorArray_Hip

Prototype `int N_VEnableWrmsNormVectorArray_Hip(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the HIP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableWrmsNormMaskVectorArray_Hip

Prototype `int N_VEnableWrmsNormMaskVectorArray_Hip(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the HIP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableScaleAddMultiVectorArray_Hip

Prototype `int N_VEnableScaleAddMultiVectorArray_Hip(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the HIP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearCombinationVectorArray_Hip

Prototype `int N_VEnableLinearCombinationVectorArray_Hip(N_Vector v,
booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the HIP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

Notes

- When there is a need to access components of an `N_Vector_Hip`, `v`, it is recommended to use functions `N_VGetDeviceArrayPointer_Hip` or `N_VGetHostArrayPointer_Hip`. However, when using managed memory, the function `N_VGetArrayPointer` may also be used.
- To maximize efficiency, vector operations in the NVECTOR_HIP 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.

**7.10.2 The SUNHipExecPolicy Class**

In order to provide maximum flexibility to users, the HIP kernel execution parameters used by kernels within SUNDIALS are defined by objects of the `sundials::HipExecPolicy` abstract class type (this class can be accessed in the global namespace as `SUNHipExecPolicy`). Thus, users may provide custom execution policies that fit the needs of their problem. The `sundials::HipExecPolicy` is defined in the header file `sundials_hip_policies.hpp`, and is as follows:

```
class HipExecPolicy
{
public:
    virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const = 0;
    virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const = 0;
    virtual hipStream_t stream() const = 0;
```

```

    virtual HipExecPolicy* clone() const = 0;
    virtual ~HipExecPolicy() {}
};

```

To define a custom execution policy, a user simply needs to create a class that inherits from the abstract class and implements the methods. The SUNDIALS provided `sundials::HipThreadDirectExecPolicy` (aka in the global namespace as `SUNHipThreadDirectExecPolicy`) class is a good example of a what a custom execution policy may look like:

```

class HipThreadDirectExecPolicy : public HipExecPolicy
{
public:
    HipThreadDirectExecPolicy(const size_t blockDim, const hipStream_t stream = 0)
        : blockDim_(blockDim), stream_(stream)
    {}

    HipThreadDirectExecPolicy(const HipThreadDirectExecPolicy& ex)
        : blockDim_(ex.blockDim_), stream_(ex.stream_)
    {}

    virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const
    {
        return (numWorkUnits + blockSize() - 1) / blockSize();
    }

    virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const
    {
        return blockDim_;
    }

    virtual hipStream_t stream() const
    {
        return stream_;
    }

    virtual HipExecPolicy* clone() const
    {
        return static_cast<HipExecPolicy*>(new HipThreadDirectExecPolicy(*this));
    }

private:
    const hipStream_t stream_;
    const size_t blockDim_;
};

```

In total, SUNDIALS provides 3 execution policies:

1. `SUNHipThreadDirectExecPolicy(const size_t blockDim, const hipStream_t stream = 0)` maps each HIP thread to a work unit. The number of threads per block (`blockDim`) can be set to anything. The grid size will be calculated so that there are enough threads for one thread per element. If a HIP stream is provided, it will be used to execute the kernel.
2. `SUNHipGridStrideExecPolicy(const size_t blockDim, const size_t gridDim, const hipStream_t stream = 0)` is for kernels that use grid stride loops. The number of threads per block (`blockDim`) can be set to anything. The number of blocks (`gridDim`) can be set to anything. If a HIP stream is provided, it will be used to execute the kernel.

3. `SUNHipBlockReduceExecPolicy(const size_t blockDim, const size_t gridDim, const hipStream_t stream = 0)` is for kernels performing a reduction across individual thread blocks. The number of threads per block (`blockDim`) can be set to any valid multiple of the HIP warp size. The grid size (`gridDim`) can be set to any value greater than 0. If it is set to 0, then the grid size will be chosen so that there is enough threads for one thread per work unit. If a HIP stream is provided, it will be used to execute the kernel.

For example, a policy that uses 128 threads per block and a user provided stream can be created like so:

```
hipStream_t stream;
hipStreamCreate(&stream);
SUNHipThreadDirectExecPolicy thread_direct(128, stream);
```

These default policy objects can be reused for multiple SUNDIALS data structures since they do not hold any modifiable state information.

7.11 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 AMD, NVIDIA, or Intel 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 either the NVIDIA CUDA programming environment, the AMD ROCm HIP programming environment, or a compiler that supports the SYCL abstraction layer. When using the AMD ROCm HIP environment, the HIP-clang compiler must be utilized. Users can select which backend to compile with by setting the `SUNDIALS_RAJA_BACKENDS` CMake variable to either `CUDA`, `HIP`, or `SYCL`. Besides the `CUDA`, `HIP`, and `SYCL` backends, RAJA has other backends such as `serial`, `OpenMP`, and `OpenACC`. These backends are not used in this SUNDIALS release.

The vector content layout is as follows:

```
struct _N_VectorContent_Raja
{
    sunindextype    length;
    booleantype     own_helper;
    SUNMemory       host_data;
    SUNMemory       device_data;
    SUNMemoryHelper mem_helper;
    void*           priv; /* 'private' data */
};
```

The content members are the vector length (size), a boolean flag that signals if the vector owns the memory helper, `SUNMemory` objects for vector data on the host and the device, a `SUNMemoryHelper` object and a private data structure which holds the memory management type, which should not be accessed directly.

When instantiated with `N_VNew_Raja`, the underlying data will be allocated on both the host and the device. Alternatively, a user can provide host and device data arrays by using the `N_VMake_Raja` constructor. To use managed memory, the constructors `N_VNewManaged_Raja` and `N_VMakeManaged_Raja` are provided. Details on each of these constructors are provided below.

The header file to include when using this module is `nvector_raja.h`. The installed module library to link to are `libsundials_nveccudaraja.lib` when using the CUDA backend, `libsundials_nvechpraja.lib` when using the HIP backend, and `libsundials_nvecsyclraja.lib` when using the SYCL backend. The extension `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

7.11.1 NVECTOR_RAJA functions

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

N_VGetHostArrayPointer_Raja

Prototype `realtype *N_VGetHostArrayPointer_Raja(N_Vector v)`

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

N_VGetDeviceArrayPointer_Raja

Prototype `realtype *N_VGetDeviceArrayPointer_Raja(N_Vector v)`

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

N_VSetHostArrayPointer_Raja

Prototype `realtype *N_VSetHostArrayPointer_Raja(N_Vector v)`

Description This function sets the pointer to the vector data on the host. The existing pointer *will not* be freed first.

N_VSetDeviceArrayPointer_Raja

Prototype `realtype *N_VSetDeviceArrayPointer_Raja(N_Vector v)`

Description This function sets pointer to the vector data on the device. The existing pointer *will not* be freed first.

N_VIsManagedMemory_Raja

Prototype `booleantype *N_VIsManagedMemory_Raja(N_Vector v)`

Description This function returns a boolean flag indicating if the vector data is allocated in managed memory or not.

The NVECTOR_RAJA module defines the implementations of all vector operations listed in Tables [7.1.1](#), [7.1.2](#), [7.1.3](#), and [7.1.4](#), 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 NVECTOR_RAJA implementation in the future. Additionally the vector operations `N_VGetArrayPointer` and `N_VSetArrayPointer` are not provided by the RAJA vector unless managed memory is used. As such, this vector cannot be used with the SUNDIALS Fortran interfaces, nor with the SUNDIALS direct solvers and preconditioners. The NVECTOR_RAJA module provides separate functions to access data on the host and on the device. It also provides methods for copying data from the host to the device and vice versa. Usage examples of NVECTOR_RAJA are provided in some example programs for CVODE [\[28\]](#).

The names of vector operations are obtained from those in Tables [7.1.1](#), [7.1.2](#), [7.1.3](#), and [7.1.4](#) by appending the suffix `_Raja` (e.g. `N_VDestroy_Raja`). The module NVECTOR_RAJA provides the following additional user-callable routines:

N_VNew_Raja

Prototype `N_Vector N_VNew_Raja(sunindextype length)`

Description This function creates and allocates memory for a RAJA `N_Vector`. The vector data array is allocated on both the host and device.

N_VNewWithMemHelp_Raja

Prototype `N_Vector N_VNewWithMemHelp_Raja(sunindextype length, booleantype use_managed_mem, SUNMemoryHelper helper);`

Description This function creates an NVECTOR_RAJA which will use the `SUNMemoryHelper` object to allocate memory. If `use_managed_memory` is 0, then unmanaged memory is used, otherwise managed memory is used.

N_VNewManaged_Raja

Prototype `N_Vector N_VNewManaged_Raja(sunindextype length)`

Description This function creates and allocates memory for a RAJA `N_Vector`. The vector data array is allocated in managed memory.

N_VNewEmpty_Raja

Prototype `N_Vector N_VNewEmpty_Raja()`

Description 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_VMake_Raja

Prototype `N_Vector N_VMake_Raja(sunindextype length, reatype *h_data, reatype *dev_data)`

Description This function creates an NVECTOR_RAJA with user-supplied vector data arrays `h_vdata` and `d_vdata`. This function does not allocate memory for data itself.

N_VMakeManaged_Raja

Prototype `N_Vector N_VMakeManaged_Raja(sunindextype length, reatype *vdata)`

Description This function creates an NVECTOR_RAJA with a user-supplied managed memory data array. This function does not allocate memory for data itself.

N_VCopyToDevice_Raja

Prototype `reatype *N_VCopyToDevice_Raja(N_Vector v)`

Description This function copies host vector data to the device.

N_VCopyFromDevice_Raja

Prototype `reatype *N_VCopyFromDevice_Raja(N_Vector v)`

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

N_VPrint_Raja

Prototype `void N_VPrint_Raja(N_Vector v)`

Description This function prints the content of a RAJA vector to `stdout`.

N_VPrintFile_Raja

Prototype `void N_VPrintFile_Raja(N_Vector v, FILE *outfile)`

Description This function prints the content of a RAJA vector to `outfile`.

By default all fused and vector array operations are disabled in the NVECTOR_RAJA module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_Raja`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone`. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VNew_Raja` will have the default settings for the NVECTOR_RAJA module.

N_VEnableFusedOps_Raja

Prototype `int N_VEnableFusedOps_Raja(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the RAJA vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

N_VEnableLinearCombination_Raja

Prototype `int N_VEnableLinearCombination_Raja(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the RAJA vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

N_VEnableScaleAddMulti_Raja

Prototype `int N_VEnableScaleAddMulti_Raja(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector to multiple vectors fused operation in the RAJA vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

N_VEnableLinearSumVectorArray_Raja

Prototype `int N_VEnableLinearSumVectorArray_Raja(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear sum operation for vector arrays in the RAJA vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

N_VEnableScaleVectorArray_Raja

Prototype `int N_VEnableScaleVectorArray_Raja(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale operation for vector arrays in the RAJA vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

N_VEnableConstVectorArray_Raja

Prototype `int N_VEnableConstVectorArray_Raja(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the const operation for vector arrays in the RAJA vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

`N_VEnableScaleAddMultiVectorArray_Raja`

Prototype `int N_VEnableScaleAddMultiVectorArray_Raja(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector array to multiple vector arrays operation in the RAJA vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

`N_VEnableLinearCombinationVectorArray_Raja`

Prototype `int N_VEnableLinearCombinationVectorArray_Raja(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination operation for vector arrays in the RAJA vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

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`. However, when using managed memory, the function `N_VGetArrayPointer` may also be used.
- To maximize efficiency, vector operations in the `NVECTOR_RAJA` implementation that have more than one `N_Vector` argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.



7.12 The NVECTOR_SYCL implementation

The `NVECTOR_SYCL` module is an experimental `NVECTOR` implementation using the `SYCL` abstraction layer. At present the only supported `SYCL` compiler is the DPC++ (Intel oneAPI) compiler. This module allows for `SUNDIALS` vector kernels to run on Intel GPU devices. The module is intended for users who are already familiar with `SYCL` and GPU programming.

The vector content layout is as follows:

```
struct _N_VectorContent_Sycl
{
    sunindextype      length;
    booleantype      own_exec;
    booleantype      own_helper;
    SUNMemory        host_data;
    SUNMemory        device_data;
    SUNSyclExecPolicy* stream_exec_policy;
    SUNSyclExecPolicy* reduce_exec_policy;
    SUNMemoryHelper  mem_helper;
    sycl::queue*     queue;
    void*            priv; /* 'private' data */
};

typedef struct _N_VectorContent_Sycl *N_VectorContent_Sycl;
```

The content members are the vector length (size), boolean flags that indicate if the vector owns the execution policies and memory helper objects (i.e., it is in charge of freeing the objects), `SUNMemory` objects for the vector data on the host and device, pointers to execution policies that control how streaming and reduction kernels are launched, a `SUNMemoryHelper` for performing memory operations, the SYCL queue, and a private data structure which holds additional members that should not be accessed directly.

When instantiated with `N_VNew_Sycl()`, the underlying data will be allocated on both the host and the device. Alternatively, a user can provide host and device data arrays by using the `N_VMake_Sycl()` constructor. To use managed (shared) memory, the constructors `N_VNewManaged_Sycl()` and `N_VMakeManaged_Sycl()` are provided. Additionally, a user-defined `SUNMemoryHelper` for allocating/freeing data can be provided with the constructor `N_VNewWithMemHelp_Sycl()`. Details on each of these constructors are provided below.

The header file to include when using this is `nvector_sycl.h`. The installed module library to link to is `libsundials_nvecsycl.lib`. The extension `.lib` is typically `.so` for shared libraries `.a` for static libraries.

7.12.1 NVECTOR_SYCL functions

The `NVECTOR_SYCL` module implementations of all vector operations listed in the sections in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4, except for `N_VDotProdMulti`, `N_VWrmsNormVectorArray`, and `N_VWrmsNormMaskVectorArray` as support for arrays of reduction vectors is not yet supported. These function will be added to the `NVECTOR_SYCL` implementation in the future. The names of vector operations are obtained from those in the aforementioned sections by appending the suffix `_Sycl` (e.g., `N_VDestroy_Sycl`).

Additionally, the `NVECTOR_SYCL` module provides the following user-callable constructors for creating a new `NVECTOR_SYCL`:

`N_VNew_Sycl`

Prototype `N_Vector N_VNew_Sycl(sunindextype length, sycl::queue* Q)`

Description This function creates and allocates memory for a SYCL `N_Vector`. The vector data array is allocated on both the host and device.

`N_VNewManaged_Sycl`

Prototype `N_Vector N_VNewManaged_Sycl(sunindextype length, sycl::queue* Q)`

Description This function creates and allocates memory for a SYCL `N_Vector`. The vector data array is allocated in managed memory.

`N_VMake_Sycl`

Prototype `N_Vector N_VMake_Sycl(sunindextype length, realtype *h_data,
realtype *dev_data, sycl::queue* Q)`

Description This function creates an `NVECTOR_SYCL` with user-supplied vector data arrays `h_vdata` and `d_vdata`. This function does not allocate memory for data itself.

`N_VMakeManaged_Sycl`

Prototype `N_Vector N_VMakeManaged_Sycl(sunindextype length, realtype *vdata,
sycl::queue* Q)`

Description This function creates an `NVECTOR_SYCL` with a user-supplied managed memory data array. This function does not allocate memory for data itself.

`N_VNewWithMemHelp_Sycl`

Prototype `N_Vector N_VNewWithMemHelp_Sycl(sunindextype length,
booleantype use_managed_mem,
SUNMemoryHelper helper, sycl::queue* Q);`

Description This function creates an NVECTOR_SYCL which will use the SUNMemoryHelper object to allocate memory. If `use_managed_memory` is 0, then unmanaged memory is used, otherwise managed memory is used.

`N_VNewEmpty_Sycl`

Prototype `N_Vector N_VNewEmpty_Sycl()`

Description This function creates a new NVECTOR_SYCL where the members of the content structure have not been allocated. This utility function is used by the other constructors to create a new vector.

The following user-callable functions are provided for accessing the vector data arrays on the host and device and copying data between the two memory spaces. Note the generic NVECTOR operations `N_VGetArrayPointer()` and `N_VSetArrayPointer()` are mapped to the corresponding `HostArray` functions given below. To ensure memory coherency, a user will need to call the `CopyTo` or `CopyFrom` functions as necessary to transfer data between the host and device, unless managed (shared) memory is used.

`N_VGetHostArrayPointer_Sycl`

Prototype `realtype *N_VGetHostArrayPointer_Sycl(N_Vector v)`

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

`N_VGetDeviceArrayPointer_Sycl`

Prototype `realtype *N_VGetDeviceArrayPointer_Sycl(N_Vector v)`

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

`N_VSetHostArrayPointer_Sycl`

Prototype `realtype *N_VSetHostArrayPointer_Sycl(N_Vector v)`

Description This function sets the pointer to the vector data on the host. The existing pointer *will not* be freed first.

`N_VSetDeviceArrayPointer_Sycl`

Prototype `realtype *N_VSetDeviceArrayPointer_Sycl(N_Vector v)`

Description This function sets pointer to the vector data on the device. The existing pointer *will not* be freed first.

`N_VCopyToDevice_Sycl`

Prototype `realtype *N_VCopyToDevice_Sycl(N_Vector v)`

Description This function copies host vector data to the device.

`N_VCopyFromDevice_Sycl`

Prototype `realtype *N_VCopyFromDevice_Sycl(N_Vector v)`

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

`N_VIsManagedMemory_Sycl`

Prototype `booleantype *N_VIsManagedMemory_Sycl(N_Vector v)`

Description This function returns a boolean flag indicating if the vector data is allocated in managed memory or not.

The following user-callable function is provided to set the execution policies for how SYCL kernels are launched on a device.

`N_VSetKernelExecPolicy_Sycl`

Prototype `int N_VSetKernelExecPolicy_Sycl(N_Vector v, SUNSyclExecPolicy *stream_exec_policy, SUNSyclExecPolicy *reduce_exec_policy)`

Description This function sets the execution policies which control the kernel parameters utilized when launching the streaming and reduction kernels. By default the vector is setup to use the `SUNSyclThreadDirectExecPolicy` and `SUNSyclBlockReduceExecPolicy`. See Section 7.12.2 below for more information about the `SUNSyclExecPolicy` class.

Note: All vectors used in a single instance of a SUNDIALS package must use the same execution policy. It is **strongly recommended** that this function is called immediately after constructing the vector, and any subsequent vector be created by cloning to ensure consistent execution policies across vectors.

The following user-callable functions are provided to print the host vector data array. Unless managed memory is used, a user may need to call `N_VCopyFromDevice_Sycl()` to ensure consistency between the host and device array.

`N_VPrint_Sycl`

Prototype `void N_VPrint_Sycl(N_Vector v)`

Description This function prints the host data of a SYCL vector to `stdout`.

`N_VPrintFile_Sycl`

Prototype `void N_VPrintFile_Sycl(N_Vector v, FILE *outfile)`

Description This function prints the host data of a SYCL vector to `outfile`.

By default all fused and vector array operations are disabled in the `NVECTOR_SYCL` module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with one of the above constructors, enable/disable the desired operations on that vector with the functions below, and then use this vector in conjunction `N_VClone` to create any additional vectors. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created by any of the above constructors will have the default settings for the `NVECTOR_SYCL` module.

`N_VEnableFusedOps_Sycl`

Prototype `int N_VEnableFusedOps_Sycl(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the SYCL vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

N_VEnableLinearCombination_Sycl

Prototype `int N_VEnableLinearCombination_Sycl(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the SYCL vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableScaleAddMulti_Sycl

Prototype `int N_VEnableScaleAddMulti_Sycl(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the SYCL vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearSumVectorArray_Sycl

Prototype `int N_VEnableLinearSumVectorArray_Sycl(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the SYCL vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableScaleVectorArray_Sycl

Prototype `int N_VEnableScaleVectorArray_Sycl(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the SYCL vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableConstVectorArray_Sycl

Prototype `int N_VEnableConstVectorArray_Sycl(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the SYCL vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableScaleAddMultiVectorArray_Sycl

Prototype `int N_VEnableScaleAddMultiVectorArray_Sycl(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the SYCL vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearCombinationVectorArray_Sycl

Prototype `int N_VEnableLinearCombinationVectorArray_Sycl(N_Vector v,
booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the SYCL vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

Notes

- When there is a need to access components of an `N_Vector_Sycl`, `v`, it is recommended to use `N_VGetDeviceArrayPointer` to access the device array or `N_VGetArrayPointer` for the host array. When using managed (shared) memory, either function may be used. To ensure memory coherency, a user may need to call the `CopyTo` or `CopyFrom` functions as necessary to transfer data between the host and device, unless managed (shared) memory is used.
- To maximize efficiency, vector operations in the `NVECTOR_SYCL` 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.



7.12.2 The SUNSyclExecPolicy Class

In order to provide maximum flexibility to users, the SYCL kernel execution parameters used by kernels within SUNDIALS are defined by objects of the `sundials::SyclExecPolicy` abstract class type (this class can be accessed in the global namespace as `SUNSyclExecPolicy`). Thus, users may provide custom execution policies that fit the needs of their problem. The `sundials::SyclExecPolicy` is defined in the header file `sundials_sycl_policies.hpp`, as follows:

```
class SyclExecPolicy
{
public:
    virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const = 0;
    virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const = 0;
    virtual SyclExecPolicy* clone() const = 0;
    virtual ~SyclExecPolicy() {}
};
```

For consistency the function names and behavior mirror the execution policies for the CUDA and HIP vectors. In the SYCL case the `blockSize` is the local work-group range in a one-dimensional `nd_range` (threads per group). The `gridSize` is the number of local work groups so the global work-group range in a one-dimensional `nd_range` is `blockSize * gridSize` (total number of threads). All vector kernels are written with a many-to-one mapping where work units (vector elements) are mapped in a round-robin manner across the global range. As such, the `blockSize` and `gridSize` can be set to any positive value.

To define a custom execution policy, a user simply needs to create a class that inherits from the abstract class and implements the methods. The SUNDIALS provided `sundials::SyclThreadDirectExecPolicy` (aka in the global namespace as `SUNSyclThreadDirectExecPolicy`) class is a good example of a what a custom execution policy may look like:

```
class SyclThreadDirectExecPolicy : public SyclExecPolicy
{
public:
    SyclThreadDirectExecPolicy(const size_t blockDim)
        : blockDim_(blockDim)
    {}

    SyclThreadDirectExecPolicy(const SyclThreadDirectExecPolicy& ex)
        : blockDim_(ex.blockDim_)
    {}

    virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const
    {
```

```

    return (numWorkUnits + blockSize() - 1) / blockSize();
}

virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const
{
    return blockDim_;
}

virtual SyclExecPolicy* clone() const
{
    return static_cast<SyclExecPolicy*>(new SyclThreadDirectExecPolicy(*this));
}

private:
    const size_t blockDim_;
};

```

SUNDIALS provides the following execution policies:

1. `SUNSyclThreadDirectExecPolicy(const size_t blockDim)` is for kernels performing streaming operations and maps each work unit (vector element) to a work-item (thread). Based on the local work-group range (number of threads per group, `blockSize`) the number of local work-groups (`gridSize`) is computed so there are enough work-items in the global work-group range (total number of threads, `blockSize * gridSize`) for one work unit per work-item (thread).
2. `SUNSyclGridStrideExecPolicy(const size_t blockDim, const size_t gridDim)` is for kernels performing streaming operations and maps each work unit (vector element) to a work-item (thread) in a round-robin manner so the local work-group range (number of threads per group, `blockSize`) and the number of local work-groups (`gridSize`) can be set to any positive value. In this case the global work-group range (total number of threads, `blockSize * gridSize`) may be less than the number of work units (vector elements).
3. `SUNSyclBlockReduceExecPolicy(const size_t blockDim)` is for kernels performing a reduction, the local work-group range (number of threads per group, `blockSize`) and the number of local work-groups (`gridSize`) can be set to any positive value or the `gridSize` may be set to 0 in which case the global range is chosen so that there are enough threads for at most two work units per work-item.

By default the `NVECTOR_SYCL` module uses the `SUNSyclThreadDirectExecPolicy` and `SUNSyclBlockReduceExecPolicy` where the default `blockDim` is determined by querying the device for the `max_work_group_size`. User may specify different policies by constructing a new `SyclExecPolicy` and attaching it with `N_VSetKernelExecPolicy_Sycl()`. For example, a policy that uses 128 work-items (threads) per group can be created and attached like so:

```

N_Vector v = N_VNew_Sycl(length);
SUNSyclThreadDirectExecPolicy thread_direct(128);
SUNSyclBlockReduceExecPolicy block_reduce(128);
flag = N_VSetKernelExecPolicy_Sycl(v, &thread_direct, &block_reduce);

```

These default policy objects can be reused for multiple SUNDIALS data structures (e.g. a `SUNMatrix` and an `N_Vector`) since they do not hold any modifiable state information.

7.13 The NVECTOR_OPENMPDEV implementation

In situations where a user has access to a device such as a GPU for offloading computation, SUNDIALS provides an NVECTOR implementation using OpenMP device offloading, called `NVECTOR_OPENMPDEV`.

The NVECTOR_OPENMPDEV implementation defines the *content* field of the `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array on the host, a pointer to the beginning of a contiguous data array on the device, and a boolean flag `own_data` which specifies the ownership of host and device data arrays.

```
struct _N_VectorContent_OpenMPDEV {
    sunindextype length;
    booleantype own_data;
    realtype *host_data;
    realtype *dev_data;
};
```

The header file to include when using this module is `nvector_openmpdev.h`. The installed module library to link to is `libsundials_nvecopenmpdev.lib` where *.lib* is typically *.so* for shared libraries and *.a* for static libraries.

7.13.1 NVECTOR_OPENMPDEV accessor macros

The following macros are provided to access the content of an NVECTOR_OPENMPDEV vector.

- `NV_CONTENT_OMPDEV`

This routine gives access to the contents of the NVECTOR_OPENMPDEV vector `N_Vector`.

The assignment `v_cont = NV_CONTENT_OMPDEV(v)` sets `v_cont` to be a pointer to the NVECTOR_OPENMPDEV `N_Vector` content structure.

Implementation:

```
#define NV_CONTENT_OMPDEV(v) ( (N_VectorContent_OpenMPDEV)(v->content) )
```

- `NV_OWN_DATA_OMPDEV`, `NV_DATA_HOST_OMPDEV`, `NV_DATA_DEV_OMPDEV`, `NV_LENGTH_OMPDEV`

These macros give individual access to the parts of the content of an NVECTOR_OPENMPDEV `N_Vector`.

The assignment `v_data = NV_DATA_HOST_OMPDEV(v)` sets `v_data` to be a pointer to the first component of the data on the host for the `N_Vector` `v`. The assignment `NV_DATA_HOST_OMPDEV(v) = v_data` sets the host component array of `v` to be `v_data` by storing the pointer `v_data`.

The assignment `v_dev_data = NV_DATA_DEV_OMPDEV(v)` sets `v_dev_data` to be a pointer to the first component of the data on the device for the `N_Vector` `v`. The assignment `NV_DATA_DEV_OMPDEV(v) = v_dev_data` sets the device component array of `v` to be `v_dev_data` by storing the pointer `v_dev_data`.

The assignment `v_len = NV_LENGTH_OMPDEV(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_OMPDEV(v) = len_v` sets the length of `v` to be `len_v`.

Implementation:

```
#define NV_OWN_DATA_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->own_data )
#define NV_DATA_HOST_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->host_data )
#define NV_DATA_DEV_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->dev_data )
#define NV_LENGTH_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->length )
```

7.13.2 NVECTOR_OPENMPDEV functions

The NVECTOR_OPENMPDEV module defines OpenMP device offloading implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4, except for `NVGetArrayPointer` and `NVSetArrayPointer`. As such, this vector cannot be used with the SUNDIALS Fortran interfaces, nor with the SUNDIALS direct solvers and preconditioners. It also provides methods for copying from the host to the device and vice versa.

The names of vector operations are obtained from those in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4 by appending the suffix `_OpenMPDEV` (e.g. `N_VDestroy_OpenMPDEV`). The module `NVECTOR_OPENMPDEV` provides the following additional user-callable routines:

`N_VNew_OpenMPDEV`

Prototype `N_Vector N_VNew_OpenMPDEV(sunindextype vec_length)`

Description This function creates and allocates memory for an `NVECTOR_OPENMPDEV` `N_Vector`.

`N_VNewEmpty_OpenMPDEV`

Prototype `N_Vector N_VNewEmpty_OpenMPDEV(sunindextype vec_length)`

Description This function creates a new `NVECTOR_OPENMPDEV` `N_Vector` with an empty (`NULL`) host and device data arrays.

`N_VMake_OpenMPDEV`

Prototype `N_Vector N_VMake_OpenMPDEV(sunindextype vec_length, realtype *h_vdata, realtype *d_vdata)`

Description This function creates an `NVECTOR_OPENMPDEV` vector with user-supplied vector data arrays `h_vdata` and `d_vdata`. This function does not allocate memory for data itself.

`N_VCloneVectorArray_OpenMPDEV`

Prototype `N_Vector *N_VCloneVectorArray_OpenMPDEV(int count, N_Vector w)`

Description This function creates (by cloning) an array of `count` `NVECTOR_OPENMPDEV` vectors.

`N_VCloneVectorArrayEmpty_OpenMPDEV`

Prototype `N_Vector *N_VCloneVectorArrayEmpty_OpenMPDEV(int count, N_Vector w)`

Description This function creates (by cloning) an array of `count` `NVECTOR_OPENMPDEV` vectors, each with an empty (`NULL`) data array.

`N_VDestroyVectorArray_OpenMPDEV`

Prototype `void N_VDestroyVectorArray_OpenMPDEV(N_Vector *vs, int count)`

Description This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_OpenMPDEV` or with `N_VCloneVectorArrayEmpty_OpenMPDEV`.

`N_VGetHostArrayPointer_OpenMPDEV`

Prototype `realtype *N_VGetHostArrayPointer_OpenMPDEV(N_Vector v)`

Description This function returns a pointer to the host data array.

`N_VGetDeviceArrayPointer_OpenMPDEV`

Prototype `realtype *N_VGetDeviceArrayPointer_OpenMPDEV(N_Vector v)`

Description This function returns a pointer to the device data array.

N_VPrint_OpenMPDEV

Prototype `void N_VPrint_OpenMPDEV(N_Vector v)`

Description This function prints the content of an NVECTOR_OPENMPDEV vector to `stdout`.

N_VPrintFile_OpenMPDEV

Prototype `void N_VPrintFile_OpenMPDEV(N_Vector v, FILE *outfile)`

Description This function prints the content of an NVECTOR_OPENMPDEV vector to `outfile`.

N_VCopyToDevice_OpenMPDEV

Prototype `void N_VCopyToDevice_OpenMPDEV(N_Vector v)`

Description This function copies the content of an NVECTOR_OPENMPDEV vector's host data array to the device data array.

N_VCopyFromDevice_OpenMPDEV

Prototype `void N_VCopyFromDevice_OpenMPDEV(N_Vector v)`

Description This function copies the content of an NVECTOR_OPENMPDEV vector's device data array to the host data array.

By default all fused and vector array operations are disabled in the NVECTOR_OPENMPDEV module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_OpenMPDEV`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone`. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with `N_VNew_OpenMPDEV` will have the default settings for the NVECTOR_OPENMPDEV module.

N_VEnableFusedOps_OpenMPDEV

Prototype `int N_VEnableFusedOps_OpenMPDEV(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

N_VEnableLinearCombination_OpenMPDEV

Prototype `int N_VEnableLinearCombination_OpenMPDEV(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

N_VEnableScaleAddMulti_OpenMPDEV

Prototype `int N_VEnableScaleAddMulti_OpenMPDEV(N_Vector v, booleantype tf)`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector to multiple vectors fused operation in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

`N_VEnableDotProdMulti_OpenMPDEV`

Prototype `int N_VEnableDotProdMulti_OpenMPDEV(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

`N_VEnableLinearSumVectorArray_OpenMPDEV`

Prototype `int N_VEnableLinearSumVectorArray_OpenMPDEV(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

`N_VEnableScaleVectorArray_OpenMPDEV`

Prototype `int N_VEnableScaleVectorArray_OpenMPDEV(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

`N_VEnableConstVectorArray_OpenMPDEV`

Prototype `int N_VEnableConstVectorArray_OpenMPDEV(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

`N_VEnableWrmsNormVectorArray_OpenMPDEV`

Prototype `int N_VEnableWrmsNormVectorArray_OpenMPDEV(N_Vector v, booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

`N_VEnableWrmsNormMaskVectorArray_OpenMPDEV`

Prototype `int N_VEnableWrmsNormMaskVectorArray_OpenMPDEV(N_Vector v,
booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

`N_VEnableScaleAddMultiVectorArray_OpenMPDEV`

Prototype `int N_VEnableScaleAddMultiVectorArray_OpenMPDEV(N_Vector v,
booleantype tf)`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

```
N_VEnableLinearCombinationVectorArray_OpenMPDEV
```

```
Prototype    int N_VEnableLinearCombinationVectorArray_OpenMPDEV(N_Vector v,
                                                             booleantype tf)
```

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

Notes

- When looping over the components of an N_Vector v, it is most efficient to first obtain the component array via `h_data = NV_DATA_HOST_OMPDEV(v)` for the host array or `d_data = NV_DATA_DEV_OMPDEV(v)` for the device array and then access `h_data[i]` or `d_data[i]` within the loop.
- When accessing individual components of an N_Vector v on the host remember to first copy the array back from the device with `N_VCopyFromDevice_OpenMPDEV(v)` to ensure the array is up to date.
- N_VNewEmpty_OpenMPDEV, N_VMake_OpenMPDEV, and N_VCloneVectorArrayEmpty_OpenMPDEV set the field `own_data = SUNFALSE`. N_VDestroy_OpenMPDEV and N_VDestroyVectorArray_OpenMPDEV 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_OPENMPDEV 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.



7.14 The NVECTOR_TRILINOS implementation

The NVECTOR_TRILINOS module is an NVECTOR wrapper around the Trilinos Tpetra vector. The interface to Tpetra is implemented in the `Sundials::TpetraVectorInterface` class. This class simply stores a reference counting pointer to a Tpetra vector and inherits from an empty structure

```
struct _N_VectorContent_Trilinos {};
```

to interface the C++ class with the NVECTOR C code. A pointer to an instance of this class is kept in the `content` field of the N_Vector object, to ensure that the Tpetra vector is not deleted for as long as the N_Vector object exists.

The Tpetra vector type in the `Sundials::TpetraVectorInterface` class is defined as:

```
typedef Tpetra::Vector<realttype, int, sunindextype> vector_type;
```

The Tpetra vector will use the SUNDIALS-specified `realttype` as its scalar type, `int` as its local ordinal type, and `sunindextype` as the global ordinal type. This type definition will use Tpetra's default node type. Available Kokkos node types in Trilinos 12.14 release are serial (single thread), OpenMP, Pthread, and CUDA. The default node type is selected when building the Kokkos package. For example, the Tpetra vector will use a CUDA node if Tpetra was built with CUDA support and the CUDA node was selected as the default when Tpetra was built.

The header file to include when using this module is `nvector_trilinos.h`. The installed module library to link to is `libsundials_nvectrilinos.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

7.14.1 NVECTOR_TRILINOS functions

The NVECTOR_TRILINOS module defines implementations of all vector operations listed in Tables 7.1.1, 7.1.4, and 7.1.4, except for N_VGetArrayPointer and N_VSetArrayPointer. As such, this vector cannot be used with SUNDIALS Fortran interfaces, nor with the SUNDIALS direct solvers and preconditioners. When access to raw vector data is needed, it is recommended to extract the Trilinos Tpetra vector first, and then use Tpetra vector methods to access the data. Usage examples of NVECTOR_TRILINOS are provided in example programs for IDA [27].

The names of vector operations are obtained from those in Tables 7.1.1, 7.1.4, and 7.1.4 by appending the suffix `_Trilinos` (e.g. `N_VDestroy_Trilinos`). Vector operations call existing `Tpetra::Vector` methods when available. Vector operations specific to SUNDIALS are implemented as standalone functions in the namespace `Sundials::TpetraVector`, located in the file `SundialsTpetraVectorKernels.hpp`. The module NVECTOR_TRILINOS provides the following additional user-callable functions:

- `N_VGetVector_Trilinos`

This C++ function takes an `N_Vector` as the argument and returns a reference counting pointer to the underlying Tpetra vector. This is a standalone function defined in the global namespace.

```
Teuchos::RCP<vector_type> N_VGetVector_Trilinos(N_Vector v);
```

- `N_VMake_Trilinos`

This C++ function creates and allocates memory for an NVECTOR_TRILINOS wrapper around a user-provided Tpetra vector. This is a standalone function defined in the global namespace.

```
N_Vector N_VMake_Trilinos(Teuchos::RCP<vector_type> v);
```

Notes

- The template parameter `vector_type` should be set as:

```
typedef Sundials::TpetraVectorInterface::vector_type vector_type
```

This will ensure that data types used in Tpetra vector match those in SUNDIALS.
- When there is a need to access components of an `N_Vector_Trilinos`, `v`, it is recommended to extract the Trilinos vector object via `x_vec = N_VGetVector_Trilinos(v)` and then access components using the appropriate Trilinos functions.
- The functions `N_VDestroy_Trilinos` and `N_VDestroyVectorArray_Trilinos` only delete the `N_Vector` wrapper. The underlying Tpetra vector object will exist for as long as there is at least one reference to it.

7.15 The NVECTOR_MANYVECTOR implementation

The NVECTOR_MANYVECTOR implementation of the NVECTOR module provided with SUNDIALS is designed to facilitate problems with an inherent data partitioning for the solution vector within a computational node. These data partitions are entirely user-defined, through construction of distinct NVECTOR modules for each component, that are then combined together to form the NVECTOR_MANYVECTOR. We envision two generic use cases for this implementation:

- *Heterogeneous computational architectures*: for users who wish to partition data on a node between different computing resources, they may create architecture-specific subvectors for each partition. For example, a user could create one serial component based on NVECTOR_SERIAL, another component for GPU accelerators based on NVECTOR_CUDA, and another threaded component based on NVECTOR_OPENMP.

Description This function creates a `ManyVector` from a set of existing `NVECTOR` objects.

This routine will copy all `N_Vector` pointers from the input `vec_array`, so the user may modify/free that pointer array after calling this function. However, this routine does *not* allocate any new subvectors, so the underlying `NVECTOR` objects themselves should not be destroyed before the `ManyVector` that contains them.

Upon successful completion, the new `ManyVector` is returned; otherwise this routine returns `NULL` (e.g., a memory allocation failure occurred).

Users of the Fortran 2003 interface to this function will first need to use the generic `N_Vector` utility functions `N_VNewVectorArray`, and `N_VSetVecAtIndexVectorArray` to create the `N_Vector*` argument. This is further explained in Chapter 5.1.3.5, and the functions are documented in Chapter 7.1.6.

F2003 Name This function is callable as `FN_VNew_ManyVector` when using the Fortran 2003 interface module.

`N_VGetSubvector_ManyVector`

Prototype `N_Vector N_VGetSubvector_ManyVector(N_Vector v, sunindextype vec_num);`

Description This function returns the `vec_num` subvector from the `NVECTOR` array.

F2003 Name This function is callable as `FN_VGetSubvector_ManyVector` when using the Fortran 2003 interface module.

`N_VGetSubvectorArrayPointer_ManyVector`

Prototype `realttype *N_VGetSubvectorArrayPointer_ManyVector(N_Vector v, sunindextype vec_num);`

Description This function returns the data array pointer for the `vec_num` subvector from the `NVECTOR` array.

If the input `vec_num` is invalid, or if the subvector does not support the `N_VGetArrayPointer` operation, then `NULL` is returned.

F2003 Name This function is callable as `FN_VGetSubvectorArrayPointer_ManyVector` when using the Fortran 2003 interface module.

`N_VSetSubvectorArrayPointer_ManyVector`

Prototype `int N_VSetSubvectorArrayPointer_ManyVector(realttype *v_data, N_Vector v, sunindextype vec_num);`

Description This function sets the data array pointer for the `vec_num` subvector from the `NVECTOR` array.

If the input `vec_num` is invalid, or if the subvector does not support the `N_VSetArrayPointer` operation, then this routine returns `-1`; otherwise it returns `0`.

F2003 Name This function is callable as `FN_VSetSubvectorArrayPointer_ManyVector` when using the Fortran 2003 interface module.

`N_VGetNumSubvectors_ManyVector`

Prototype `sunindextype N_VGetNumSubvectors_ManyVector(N_Vector v);`

Description This function returns the overall number of subvectors in the `ManyVector` object.

F2003 Name This function is callable as `FN_VGetNumSubvectors_ManyVector` when using the Fortran 2003 interface module.

By default all fused and vector array operations are disabled in the NVECTOR_MANYVECTOR module, except for `N_VWrmsNormVectorArray` and `N_VWrmsNormMaskVectorArray`, that are enabled by default. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNewManyVector`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone`. This guarantees that the new vectors will have the same operations enabled/disabled, since cloned vectors inherit those configuration options from the vector they are cloned from, while vectors created with `N_VNewManyVector` will have the default settings for the NVECTOR_MANYVECTOR module. We note that these routines *do not* call the corresponding routines on subvectors, so those should be set up as desired *before* attaching them to the ManyVector in `N_VNewManyVector`.

`N_VEnableFusedOps_ManyVector`

Prototype `int N_VEnableFusedOps_ManyVector(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableFusedOps_ManyVector` when using the Fortran 2003 interface module.

`N_VEnableLinearCombination_ManyVector`

Prototype `int N_VEnableLinearCombination_ManyVector(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableLinearCombination_ManyVector` when using the Fortran 2003 interface module.

`N_VEnableScaleAddMulti_ManyVector`

Prototype `int N_VEnableScaleAddMulti_ManyVector(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableScaleAddMulti_ManyVector` when using the Fortran 2003 interface module.

`N_VEnableDotProdMulti_ManyVector`

Prototype `int N_VEnableDotProdMulti_ManyVector(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableDotProdMulti_ManyVector` when using the Fortran 2003 interface module.

N_VEnableLinearSumVectorArray_ManyVector

Prototype `int N_VEnableLinearSumVectorArray_ManyVector(N_Vector v, boolean_t tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableLinearSumVectorArray_ManyVector when using the Fortran 2003 interface module.

N_VEnableScaleVectorArray_ManyVector

Prototype `int N_VEnableScaleVectorArray_ManyVector(N_Vector v, boolean_t tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableScaleVectorArray_ManyVector when using the Fortran 2003 interface module.

N_VEnableConstVectorArray_ManyVector

Prototype `int N_VEnableConstVectorArray_ManyVector(N_Vector v, boolean_t tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableConstVectorArray_ManyVector when using the Fortran 2003 interface module.

N_VEnableWrmsNormVectorArray_ManyVector

Prototype `int N_VEnableWrmsNormVectorArray_ManyVector(N_Vector v, boolean_t tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableWrmsNormVectorArray_ManyVector when using the Fortran 2003 interface module.

N_VEnableWrmsNormMaskVectorArray_ManyVector

Prototype `int N_VEnableWrmsNormMaskVectorArray_ManyVector(N_Vector v, boolean_t tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableWrmsNormMaskVectorArray_ManyVector when using the Fortran 2003 interface module.

Notes

- N_VNew_ManyVector sets the field `own_data = SUNFALSE`. N_VDestroy_ManyVector will not attempt to call N_VDestroy on any subvectors contained in the subvector array for any N_Vector with `own_data` set to `SUNFALSE`. In such a case, it is the user's responsibility to deallocate the subvectors.





- To maximize efficiency, arithmetic vector operations in the NVECTOR_MANYVECTOR 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 subvector representations.

7.16 The NVECTOR_MPIMANYVECTOR implementation

The NVECTOR_MPIMANYVECTOR implementation of the NVECTOR module provided with SUNDIALS is designed to facilitate problems with an inherent data partitioning for the solution vector, and when using distributed-memory parallel architectures. As such, the MPIManyVector implementation supports all use cases allowed by the MPI-unaware ManyVector implementation, as well as partitioning data between nodes in a parallel environment. These data partitions are entirely user-defined, through construction of distinct NVECTOR modules for each component, that are then combined together to form the NVECTOR_MPIMANYVECTOR. We envision three generic use cases for this implementation:

- Heterogeneous computational architectures (single-node or multi-node)*: for users who wish to partition data on a node between different computing resources, they may create architecture-specific subvectors for each partition. For example, a user could create one MPI-parallel component based on NVECTOR_PARALLEL, another single-node component for GPU accelerators based on NVECTOR_CUDA, and another threaded single-node component based on NVECTOR_OPENMP.
- Process-based multiphysics decompositions (multi-node)*: for users who wish to combine separate simulations together, e.g., where one subvector resides on one subset of MPI processes, while another subvector resides on a different subset of MPI processes, and where the user has created a MPI *intercommunicator* to connect these distinct process sets together.
- Structure of arrays (SOA) data layouts (single-node or multi-node)*: for users who wish to create separate subvectors for each solution component, e.g., in a Navier-Stokes simulation they could have separate subvectors for density, velocities and pressure, which are combined together into a single NVECTOR_MPIMANYVECTOR for the overall “solution”.

We note that the above use cases are not mutually exclusive, and the NVECTOR_MPIMANYVECTOR implementation should support arbitrary combinations of these cases.

The NVECTOR_MPIMANYVECTOR implementation is designed to work with any NVECTOR subvectors that implement the minimum *required* set of operations, however significant performance benefits may be obtained when subvectors additionally implement the optional local reduction operations listed in Table 7.1.4.

Additionally, NVECTOR_MPIMANYVECTOR sets no limit on the number of subvectors that may be attached (aside from the limitations of using `sunindextype` for indexing, and standard per-node memory limitations). However, while this ostensibly supports subvectors with one entry each (i.e., one subvector for each solution entry), we anticipate that this extreme situation will hinder performance due to non-stride-one memory accesses and increased function call overhead. We therefore recommend a relatively coarse partitioning of the problem, although actual performance will likely be problem-dependent.

As a final note, in the coming years we plan to introduce additional algebraic solvers and time integration modules that will leverage the problem partitioning enabled by NVECTOR_MPIMANYVECTOR. However, even at present we anticipate that users will be able to leverage such data partitioning in their problem-defining ODE right-hand side, DAE residual, or nonlinear solver residual functions.

7.16.1 NVECTOR_MPIMANYVECTOR structure

The NVECTOR_MPIMANYVECTOR implementation defines the *content* field of N_Vector to be a structure containing the MPI communicator (or MPI_COMM_NULL if running on a single-node), the number of subvectors comprising the MPIManyVector, the global length of the MPIManyVector (including all subvectors on all MPI tasks), a pointer to the beginning of the array of subvectors, and a boolean flag `own_data` indicating ownership of the subvectors that populate `subvec.array`.

```

struct _N_VectorContent_MPIManyVector {
  MPI_Comm      comm;          /* overall MPI communicator      */
  sunindextype  num_subvectors; /* number of vectors attached   */
  sunindextype  global_length; /* overall mpimanyvector length */
  N_Vector*     subvec_array;  /* pointer to N_Vector array    */
  boolean_type  own_data;      /* flag indicating data ownership */
};

```

The header file to include when using this module is `nvector_mpimanyvector.h`. The installed module library to link against is `libsundials_nvecmpimanyvector.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

Note: If SUNDIALS is configured with MPI disabled, then the MPIManyVector library will not be built. Furthermore, any user codes that include `nvector_mpimanyvector.h` *must* be compiled using an MPI-aware compiler (whether the specific user code utilizes MPI or not). We note that the NVECTOR_MANYVECTOR implementation is designed for ManyVector use cases in an MPI-unaware environment.



7.16.2 NVECTOR_MPIMANYVECTOR functions

The NVECTOR_MPIMANYVECTOR module implements all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4, except for `N_VGetArrayPointer`, `N_VSetArrayPointer`, `N_VScaleAddMultiVectorArray`, and `N_VLinearCombinationVectorArray`. As such, this vector cannot be used with the SUNDIALS Fortran-77 interfaces, nor with the SUNDIALS direct solvers and preconditioners. Instead, the NVECTOR_MPIMANYVECTOR module provides functions to access subvectors, whose data may in turn be accessed according to their NVECTOR implementations.

The names of vector operations are obtained from those in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4 by appending the suffix `_MPIManyVector` (e.g. `N_VDestroy_MPIManyVector`). The module NVECTOR_MPIMANYVECTOR provides the following additional user-callable routines:

N_VNew_MPIManyVector

Prototype `N_Vector N_VNew_MPIManyVector(sunindextype num_subvectors, N_Vector *vec_array);`

Description This function creates an MPIManyVector from a set of existing NVECTOR objects, under the requirement that all MPI-aware subvectors use the same MPI communicator (this is checked internally). If none of the subvectors are MPI-aware, then this may equivalently be used to describe data partitioning within a single node. We note that this routine is designed to support use cases A and C above.

This routine will copy all `N_Vector` pointers from the input `vec_array`, so the user may modify/free that pointer array after calling this function. However, this routine does *not* allocate any new subvectors, so the underlying NVECTOR objects themselves should not be destroyed before the MPIManyVector that contains them.

Upon successful completion, the new MPIManyVector is returned; otherwise this routine returns NULL (e.g., if two MPI-aware subvectors use different MPI communicators).

Users of the Fortran 2003 interface to this function will first need to use the generic `N_Vector` utility functions `N_VNewVectorArray`, and `N_VSetVecAtIndexVectorArray` to create the `N_Vector*` argument. This is further explained in Chapter 5.1.3.5, and the functions are documented in Chapter 7.1.6.

F2003 Name This function is callable as `FN_VNew_MPIManyVector` when using the Fortran 2003 interface module.

N_VMake_MPIManyVector

Prototype `N_Vector N_VMake_MPIManyVector(MPI_Comm comm, sunindextype num_subvectors, N_Vector *vec_array);`

Description This function creates an MPIManyVector from a set of existing NVECTOR objects, and a user-created MPI communicator that “connects” these subvectors. Any MPI-aware subvectors may use different MPI communicators than the input `comm`. We note that this routine is designed to support any combination of the use cases above.

The input `comm` should be this user-created MPI communicator. This routine will internally call `MPI_Comm_dup` to create a copy of the input `comm`, so the user-supplied `comm` argument need not be retained after the call to `N_VMake_MPIManyVector`.

If all subvectors are MPI-unaware, then the input `comm` argument should be `MPI_COMM_NULL`, although in this case, it would be simpler to call `N_VNew_MPIManyVector` instead, or to just use the `NVECTOR_MANYVECTOR` module.

This routine will copy all `N_Vector` pointers from the input `vec_array`, so the user may modify/free that pointer array after calling this function. However, this routine does *not* allocate any new subvectors, so the underlying NVECTOR objects themselves should not be destroyed before the MPIManyVector that contains them.

Upon successful completion, the new MPIManyVector is returned; otherwise this routine returns `NULL` (e.g., if the input `vec_array` is `NULL`).

F2003 Name This function is callable as `FN_VMake_MPIManyVector` when using the Fortran 2003 interface module.

N_VGetSubvector_MPIManyVector

Prototype `N_Vector N_VGetSubvector_MPIManyVector(N_Vector v, sunindextype vec_num);`

Description This function returns the `vec_num` subvector from the NVECTOR array.

F2003 Name This function is callable as `FN_VGetSubvector_MPIManyVector` when using the Fortran 2003 interface module.

N_VGetSubvectorArrayPointer_MPIManyVector

Prototype `realtype *N_VGetSubvectorArrayPointer_MPIManyVector(N_Vector v, sunindextype vec_num);`

Description This function returns the data array pointer for the `vec_num` subvector from the NVECTOR array.

If the input `vec_num` is invalid, or if the subvector does not support the `N_VGetArrayPointer` operation, then `NULL` is returned.

F2003 Name This function is callable as `FN_VGetSubvectorArrayPointer_MPIManyVector` when using the Fortran 2003 interface module.

N_VSetSubvectorArrayPointer_MPIManyVector

Prototype `int N_VSetSubvectorArrayPointer_MPIManyVector(realtype *v_data, N_Vector v, sunindextype vec_num);`

Description This function sets the data array pointer for the `vec_num` subvector from the NVECTOR array.

If the input `vec_num` is invalid, or if the subvector does not support the `N_VSetArrayPointer` operation, then this routine returns `-1`; otherwise it returns `0`.

F2003 Name This function is callable as `FN_VSetSubvectorArrayPointer_MPIManyVector` when using the Fortran 2003 interface module.

N_VGetNumSubvectors_MPIManyVector

Prototype `sunindextype N_VGetNumSubvectors_MPIManyVector(N_Vector v);`

Description This function returns the overall number of subvectors in the MPIManyVector object.

F2003 Name This function is callable as `FN_VGetNumSubvectors_MPIManyVector` when using the Fortran 2003 interface module.

By default all fused and vector array operations are disabled in the NVECTOR_MPIMANYVECTOR module, except for `N_VWrmsNormVectorArray` and `N_VWrmsNormMaskVectorArray`, that are enabled by default. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_MPIManyVector` or `N_VMake_MPIManyVector`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone`. This guarantees that the new vectors will have the same operations enabled/disabled, since cloned vectors inherit those configuration options from the vector they are cloned from, while vectors created with `N_VNew_MPIManyVector` and `N_VMake_MPIManyVector` will have the default settings for the NVECTOR_MPIMANYVECTOR module. We note that these routines *do not* call the corresponding routines on subvectors, so those should be set up as desired *before* attaching them to the MPIManyVector in `N_VNew_MPIManyVector` or `N_VMake_MPIManyVector`.

N_VEnableFusedOps_MPIManyVector

Prototype `int N_VEnableFusedOps_MPIManyVector(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the MPIManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableFusedOps_MPIManyVector` when using the Fortran 2003 interface module.

N_VEnableLinearCombination_MPIManyVector

Prototype `int N_VEnableLinearCombination_MPIManyVector(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the MPIManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableLinearCombination_MPIManyVector` when using the Fortran 2003 interface module.

N_VEnableScaleAddMulti_MPIManyVector

Prototype `int N_VEnableScaleAddMulti_MPIManyVector(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the MPIManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableScaleAddMulti_MPIManyVector` when using the Fortran 2003 interface module.

N_VEnableDotProdMulti_MPIManyVector

Prototype `int N_VEnableDotProdMulti_MPIManyVector(N_Vector v, booleantype tf);`

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the MPIManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as `FN_VEnableDotProdMulti_MPIManyVector` when using the Fortran 2003 interface module.

`N_VEnableLinearSumVectorArray_MPIManyVector`

Prototype `int N_VEnableLinearSumVectorArray_MPIManyVector(N_Vector v, booleantype tf);`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear sum operation for vector arrays in the `MPIManyVector`. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

F2003 Name This function is callable as `FN_VEnableLinearSumVectorArray_MPIManyVector` when using the Fortran 2003 interface module.

`N_VEnableScaleVectorArray_MPIManyVector`

Prototype `int N_VEnableScaleVectorArray_MPIManyVector(N_Vector v, booleantype tf);`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale operation for vector arrays in the `MPIManyVector`. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

F2003 Name This function is callable as `FN_VEnableScaleVectorArray_MPIManyVector` when using the Fortran 2003 interface module.

`N_VEnableConstVectorArray_MPIManyVector`

Prototype `int N_VEnableConstVectorArray_MPIManyVector(N_Vector v, booleantype tf);`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the const operation for vector arrays in the `MPIManyVector`. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

F2003 Name This function is callable as `FN_VEnableConstVectorArray_MPIManyVector` when using the Fortran 2003 interface module.

`N_VEnableWrmsNormVectorArray_MPIManyVector`

Prototype `int N_VEnableWrmsNormVectorArray_MPIManyVector(N_Vector v, booleantype tf);`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the WRMS norm operation for vector arrays in the `MPIManyVector`. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

F2003 Name This function is callable as `FN_VEnableWrmsNormVectorArray_MPIManyVector` when using the Fortran 2003 interface module.

`N_VEnableWrmsNormMaskVectorArray_MPIManyVector`

Prototype `int N_VEnableWrmsNormMaskVectorArray_MPIManyVector(N_Vector v, booleantype tf);`

Description This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the masked WRMS norm operation for vector arrays in the `MPIManyVector`. The return value is 0 for success and -1 if the input vector or its `ops` structure are `NULL`.

F2003 Name This function is callable as `FN_VEnableWrmsNormMaskVectorArray_MPIManyVector` when using the Fortran 2003 interface module.

Notes



- `N_VNew_MPIManyVector` and `N_VMake_MPIManyVector` set the field `own_data = SUNFALSE`. `N_VDestroy_MPIManyVector` will not attempt to call `N_VDestroy` on any subvectors contained in the subvector array for any `N_Vector` with `own_data` set to `SUNFALSE`. In such a case, it is the user's responsibility to deallocate the subvectors.
- To maximize efficiency, arithmetic vector operations in the `NVECTOR_MPIMANYVECTOR` 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 subvector representations.



7.17 The NVECTOR_MPIPLUSX implementation

The `NVECTOR_MPIPLUSX` implementation of the `NVECTOR` module provided with `SUNDIALS` is designed to facilitate the MPI+X paradigm, where X is some form of on-node (local) parallelism (e.g. OpenMP, CUDA). This paradigm is becoming increasingly popular with the rise of heterogeneous computing architectures.

The `NVECTOR_MPIPLUSX` implementation is designed to work with any `NVECTOR` that implements the minimum *required* set of operations. However, it is not recommended to use the `NVECTOR_PARALLEL`, `NVECTOR_PARHYP`, `NVECTOR_PETSC`, or `NVECTOR_TRILINOS` implementations underneath the `NVECTOR_MPIPLUSX` module since they already provide MPI capabilities.

7.17.1 NVECTOR_MPIPLUSX structure

The `NVECTOR_MPIPLUSX` implementation is a thin wrapper around the `NVECTOR_MPIMANYVECTOR`. Accordingly, it adopts the same content structure as defined in Section 7.16.1.

The header file to include when using this module is `nvector_mpiplusx.h`. The installed module library to link against is `libsundials_nvecmpiplusx.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

Note: If `SUNDIALS` is configured with MPI disabled, then the `mpiplusx` library will not be built. Furthermore, any user codes that include `nvector_mpiplusx.h` *must* be compiled using an MPI-aware compiler.



7.17.2 NVECTOR_MPIPLUSX functions

The `NVECTOR_MPIPLUSX` module adopts all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4, from the `NVECTOR_MPIMANYVECTOR` (see section 7.16.2) except for `N_VGetArrayPointer` and `N_VSetArrayPointer`; the module provides its own implementation of these functions that call the local vector implementations. Therefore, the `NVECTOR_MPIPLUSX` module implements all of the operations listed in the referenced sections except for `N_VScaleAddMultiVectorArray`, and `N_VLinearCombinationVectorArray`. Accordingly, its compatibility with the `SUNDIALS` Fortran-77 interface, and with the `SUNDIALS` direct solvers and preconditioners depends on the local vector implementation.

The module `NVECTOR_MPIPLUSX` provides the following additional user-callable routines:

`N_VMake_MPIPlusX`

Prototype `N_Vector N_VMake_MPIPlusX(MPI_Comm comm, N_Vector *local_vector);`

Description This function creates an `MPIPlusX` vector from an existing local (i.e. on-node) `NVECTOR` object, and a user-created MPI communicator.

The input `comm` should be this user-created MPI communicator. This routine will internally call `MPI_Comm_dup` to create a copy of the input `comm`, so the user-supplied `comm` argument need not be retained after the call to `N_VMake_MPIPlusX`.

This routine will copy the `N_Vector` pointer to the input `local_vector`, so the underlying local NVECTOR object should not be destroyed before the `mpiplusx` that contains it.

Upon successful completion, the new `MPIPlusX` is returned; otherwise this routine returns `NULL` (e.g., if the input `local_vector` is `NULL`).

F2003 Name This function is callable as `FN_VMake_MPIPlusX` when using the Fortran 2003 interface module.

`N_VGetLocalVector_MPIPlusX`

Prototype `N_Vector N_VGetLocalVector_MPIPlusX(N_Vector v);`

Description This function returns the local vector underneath the the `MPIPlusX` NVECTOR.

F2003 Name This function is callable as `FN_VGetLocalVector_MPIPlusX` when using the Fortran 2003 interface module.

`N_VGetArrayPointer_MPIPlusX`

Prototype `realtype* N_VGetLocalVector_MPIPlusX(N_Vector v);`

Description This function returns the data array pointer for the local vector if the local vector implements the `N_VGetArrayPointer` operation; otherwise it returns `NULL`.

F2003 Name This function is callable as `FN_VGetArrayPointer_MPIPlusX` when using the Fortran 2003 interface module.

`N_VSetArrayPointer_MPIPlusX`

Prototype `void N_VSetArrayPointer_MPIPlusX(realtype *data, N_Vector v);`

Description This function sets the data array pointer for the local vector if the local vector implements the `N_VSetArrayPointer` operation.

F2003 Name This function is callable as `FN_VSetArrayPointer_MPIPlusX` when using the Fortran 2003 interface module.

The `NVECTOR_MPIPLUSX` module does not implement any fused or vector array operations. Instead users should enable/disable fused operations on the local vector.

Notes



- `N_VMake_MPIPlusX` sets the field `own_data = SUNFALSE`. and `N_VDestroy_MPIPlusX` will not call `N_VDestroy` on the local vector. In this case, it is the user's responsibility to deallocate the local vector.



- To maximize efficiency, arithmetic vector operations in the `NVECTOR_MPIPLUSX` 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 local vector representations.

7.18 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_VGetLength`: Compares self-reported length to calculated length.
- `Test_N_VGetCommunicator`: Compares self-reported communicator to the one used in constructor; or for MPI-unaware vectors it ensures that NULL is reported.
- `Test_N_VLinearSum` Case 1a: Test $y = x + y$
- `Test_N_VLinearSum` Case 1b: Test $y = -x + y$
- `Test_N_VLinearSum` Case 1c: Test $y = ax + y$
- `Test_N_VLinearSum` Case 2a: Test $x = x + y$
- `Test_N_VLinearSum` Case 2b: Test $x = x - y$
- `Test_N_VLinearSum` Case 2c: Test $x = x + by$
- `Test_N_VLinearSum` Case 3: Test $z = x + y$
- `Test_N_VLinearSum` Case 4a: Test $z = x - y$
- `Test_N_VLinearSum` Case 4b: Test $z = -x + y$
- `Test_N_VLinearSum` Case 5a: Test $z = x + by$
- `Test_N_VLinearSum` Case 5b: Test $z = ax + y$
- `Test_N_VLinearSum` Case 6a: Test $z = -x + by$
- `Test_N_VLinearSum` Case 6b: Test $z = ax - y$
- `Test_N_VLinearSum` Case 7: Test $z = a(x + y)$
- `Test_N_VLinearSum` Case 8: Test $z = a(x - y)$
- `Test_N_VLinearSum` Case 9: Test $z = ax + by$
- `Test_N_VConst`: Fill vector with constant and check result.
- `Test_N_VProd`: Test vector multiply: $z = x * y$
- `Test_N_VDiv`: Test vector division: $z = x / y$
- `Test_N_VScale`: Case 1: scale: $x = cx$
- `Test_N_VScale`: Case 2: copy: $z = x$
- `Test_N_VScale`: Case 3: negate: $z = -x$
- `Test_N_VScale`: Case 4: combination: $z = cx$
- `Test_N_VAbs`: Create absolute value of vector.

- `Test_N_VAddConst`: add constant vector: $z = c + x$
- `Test_N_VDotProd`: Calculate dot product of two vectors.
- `Test_N_VMaxNorm`: Create vector with known values, find and validate the max norm.
- `Test_N_VWrmsNorm`: Create vector of known values, find and validate the weighted root mean square.
- `Test_N_VWrmsNormMask`: Create vector of known values, find and validate the weighted root mean square using all elements except one.
- `Test_N_VMin`: Create vector, find and validate the min.
- `Test_N_VWL2Norm`: Create vector, find and validate the weighted Euclidean L2 norm.
- `Test_N_VL1Norm`: Create vector, find and validate the L1 norm.
- `Test_N_VCompare`: Compare vector with constant returning and validating comparison vector.
- `Test_N_VInvTest`: Test $z[i] = 1 / x[i]$
- `Test_N_VConstrMask`: Test mask of vector x with vector c .
- `Test_N_VMinQuotient`: Fill two vectors with known values. Calculate and validate minimum quotient.
- `Test_N_VLinearCombination Case 1a`: Test $x = a x$
- `Test_N_VLinearCombination Case 1b`: Test $z = a x$
- `Test_N_VLinearCombination Case 2a`: Test $x = a x + b y$
- `Test_N_VLinearCombination Case 2b`: Test $z = a x + b y$
- `Test_N_VLinearCombination Case 3a`: Test $x = x + a y + b z$
- `Test_N_VLinearCombination Case 3b`: Test $x = a x + b y + c z$
- `Test_N_VLinearCombination Case 3c`: Test $w = a x + b y + c z$
- `Test_N_VScaleAddMulti Case 1a`: $y = a x + y$
- `Test_N_VScaleAddMulti Case 1b`: $z = a x + y$
- `Test_N_VScaleAddMulti Case 2a`: $Y[i] = c[i] x + Y[i]$, $i = 1,2,3$
- `Test_N_VScaleAddMulti Case 2b`: $Z[i] = c[i] x + Y[i]$, $i = 1,2,3$
- `Test_N_VDotProdMulti Case 1`: Calculate the dot product of two vectors
- `Test_N_VDotProdMulti Case 2`: Calculate the dot product of one vector with three other vectors in a vector array.
- `Test_N_VLinearSumVectorArray Case 1`: $z = a x + b y$
- `Test_N_VLinearSumVectorArray Case 2a`: $Z[i] = a X[i] + b Y[i]$
- `Test_N_VLinearSumVectorArray Case 2b`: $X[i] = a X[i] + b Y[i]$
- `Test_N_VLinearSumVectorArray Case 2c`: $Y[i] = a X[i] + b Y[i]$
- `Test_N_VScaleVectorArray Case 1a`: $y = c y$
- `Test_N_VScaleVectorArray Case 1b`: $z = c y$

- Test_N_VScaleVectorArray Case 2a: $Y[i] = c[i] Y[i]$
- Test_N_VScaleVectorArray Case 2b: $Z[i] = c[i] Y[i]$
- Test_N_VScaleVectorArray Case 1a: $z = c$
- Test_N_VScaleVectorArray Case 1b: $Z[i] = c$
- Test_N_VWrmsNormVectorArray Case 1a: Create a vector of know values, find and validate the weighted root mean square norm.
- Test_N_VWrmsNormVectorArray Case 1b: Create a vector array of three vectors of know values, find and validate the weighted root mean square norm of each.
- Test_N_VWrmsNormMaskVectorArray Case 1a: Create a vector of know 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 know values, find and validate the weighted root mean square norm of each using all elements except one.
- Test_N_VScaleAddMultiVectorArray Case 1a: $y = a x + y$
- Test_N_VScaleAddMultiVectorArray Case 1b: $z = a x + y$
- Test_N_VScaleAddMultiVectorArray Case 2a: $Y[j][0] = a[j] X[0] + Y[j][0]$
- Test_N_VScaleAddMultiVectorArray Case 2b: $Z[j][0] = a[j] X[0] + Y[j][0]$
- Test_N_VScaleAddMultiVectorArray Case 3a: $Y[0][i] = a[0] X[i] + Y[0][i]$
- Test_N_VScaleAddMultiVectorArray Case 3b: $Z[0][i] = a[0] X[i] + Y[0][i]$
- Test_N_VScaleAddMultiVectorArray Case 4a: $Y[j][i] = a[j] X[i] + Y[j][i]$
- Test_N_VScaleAddMultiVectorArray Case 4b: $Z[j][i] = a[j] X[i] + Y[j][i]$
- Test_N_VLinearCombinationVectorArray Case 1a: $x = a x$
- Test_N_VLinearCombinationVectorArray Case 1b: $z = a x$
- Test_N_VLinearCombinationVectorArray Case 2a: $x = a x + b y$
- Test_N_VLinearCombinationVectorArray Case 2b: $z = a x + b y$
- Test_N_VLinearCombinationVectorArray Case 3a: $x = a x + b y + c z$
- Test_N_VLinearCombinationVectorArray Case 3b: $w = a x + b y + c z$
- Test_N_VLinearCombinationVectorArray Case 4a: $X[0][i] = c[0] X[0][i]$
- Test_N_VLinearCombinationVectorArray Case 4b: $Z[i] = c[0] X[0][i]$
- Test_N_VLinearCombinationVectorArray Case 5a: $X[0][i] = c[0] X[0][i] + c[1] X[1][i]$
- Test_N_VLinearCombinationVectorArray Case 5b: $Z[i] = c[0] X[0][i] + c[1] X[1][i]$
- Test_N_VLinearCombinationVectorArray Case 6a: $X[0][i] = X[0][i] + c[1] X[1][i] + c[2] X[2][i]$
- Test_N_VLinearCombinationVectorArray Case 6b: $X[0][i] = c[0] X[0][i] + c[1] X[1][i] + c[2] X[2][i]$
- Test_N_VLinearCombinationVectorArray Case 6c: $Z[i] = c[0] X[0][i] + c[1] X[1][i] + c[2] X[2][i]$

- **Test_N_VDotProdLocal**: Calculate MPI task-local portion of the dot product of two vectors.
- **Test_N_VMaxNormLocal**: Create vector with known values, find and validate the MPI task-local portion of the max norm.
- **Test_N_VMinLocal**: Create vector, find and validate the MPI task-local min.
- **Test_N_VL1NormLocal**: Create vector, find and validate the MPI task-local portion of the L1 norm.
- **Test_N_VWSqrSumLocal**: Create vector of known values, find and validate the MPI task-local portion of the weighted squared sum of two vectors.
- **Test_N_VWSqrSumMaskLocal**: Create vector of known values, find and validate the MPI task-local portion of the weighted squared sum of two vectors, using all elements except one.
- **Test_N_VInvTestLocal**: Test the MPI task-local portion of $z[i] = 1 / x[i]$
- **Test_N_VConstrMaskLocal**: Test the MPI task-local portion of the mask of vector x with vector c.
- **Test_N_VMinQuotientLocal**: Fill two vectors with known values. Calculate and validate the MPI task-local minimum quotient.

Chapter 8

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.

8.1 The SUNMatrix API

The SUNMATRIX API can be grouped into two sets of functions: the core matrix operations, and utility functions. Section 8.1.1 lists the core operations, while Section 8.1.2 lists the utility functions.

8.1.1 SUNMatrix core functions

The generic `SUNMatrix` object defines the following set of core operations:

`SUNMatGetID`

Call `id = SUNMatGetID(A);`

Description Returns the type identifier for the matrix `A`. It is used to determine the matrix implementation type (e.g. dense, banded, sparse, ...) from the abstract `SUNMatrix` interface. This is used to assess compatibility with SUNDIALS-provided linear solver implementations.

Arguments `A` (`SUNMatrix`) a SUNMATRIX object

Return value A `SUNMATRIX_ID`, possible values are given in the Table 8.2.

F2003 Name `FSUNMatGetID`

`SUNMatClone`

Call `B = SUNMatClone(A);`

Description Creates a new `SUNMatrix` of the same type as an existing matrix `A` and sets the *ops* field. It does not copy the matrix, but rather allocates storage for the new matrix.

Arguments `A` (`SUNMatrix`) a SUNMATRIX object

Return value `SUNMatrix`

F2003 Name `FSUNMatClone`

F2003 Call `type(SUNMatrix), pointer :: B`
`B => FSUNMatClone(A)`

SUNMatDestroy

Call `SUNMatDestroy(A);`

Description Destroys `A` and frees memory allocated for its internal data.

Arguments `A (SUNMatrix)` a `SUNMATRIX` object

Return value `None`

F2003 Name `FSUNMatDestroy`

SUNMatSpace

Call `ier = SUNMatSpace(A, &lrw, &liw);`

Description Returns the storage requirements for the matrix `A`. `lrw` is a `long int` containing the number of realtype words and `liw` is a `long int` containing the number of integer words.

Arguments `A (SUNMatrix)` a `SUNMATRIX` object
`lrw (sunindextype*)` the number of realtype words
`liw (sunindextype*)` the number of integer words

Return value `None`

Notes 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.

F2003 Name `FSUNMatSpace`

F2003 Call `integer(c_long) :: lrw(1), liw(1)`
`ier = FSUNMatSpace(A, lrw, liw)`

SUNMatZero

Call `ier = SUNMatZero(A);`

Description Performs the operation $A_{ij} = 0$ for all entries of the matrix `A`.

Arguments `A (SUNMatrix)` a `SUNMATRIX` object

Return value A `SUNMATRIX` return code of type `int` denoting success/failure

F2003 Name `FSUNMatZero`

SUNMatCopy

Call `ier = SUNMatCopy(A,B);`

Description Performs the operation $B_{ij} = A_{i,j}$ for all entries of the matrices `A` and `B`.

Arguments `A (SUNMatrix)` a `SUNMATRIX` object
`B (SUNMatrix)` a `SUNMATRIX` object

Return value A `SUNMATRIX` return code of type `int` denoting success/failure

F2003 Name `FSUNMatCopy`

SUNMatScaleAdd

Call `ier = SUNMatScaleAdd(c, A, B);`

Description Performs the operation $A = cA + B$.

Arguments `c` (**realtype**) constant that scales A
`A` (**SUNMatrix**) a SUNMATRIX object
`B` (**SUNMatrix**) a SUNMATRIX object

Return value A SUNMATRIX return code of type `int` denoting success/failure

F2003 Name `FSUNMatScaleAdd`

SUNMatScaleAddI

Call `ier = SUNMatScaleAddI(c, A);`

Description Performs the operation $A = cA + I$.

Arguments `c` (**realtype**) constant that scales A
`A` (**SUNMatrix**) a SUNMATRIX object

Return value A SUNMATRIX return code of type `int` denoting success/failure

F2003 Name `FSUNMatScaleAddI`

SUNMatMatvecSetup

Call `ier = SUNMatMatvecSetup(A);`

Description Performs any setup necessary to perform a matrix-vector product. It is useful for SUNMatrix implementations which need to prepare the matrix itself, or communication structures before performing the matrix-vector product.

Arguments `A` (**SUNMatrix**) a SUNMATRIX object

Return value A SUNMATRIX return code of type `int` denoting success/failure

F2003 Name `FSUNMatMatvecSetup`

SUNMatMatvec

Call `ier = SUNMatMatvec(A, x, y);`

Description 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.

Arguments `A` (**SUNMatrix**) a SUNMATRIX object
`x` (**N_Vector**) a NVECTOR object
`y` (**N_Vector**) an output NVECTOR object

Return value A SUNMATRIX return code of type `int` denoting success/failure

F2003 Name `FSUNMatMatvec`

8.1.2 SUNMatrix utility functions

To aid in the creation of custom SUNMATRIX modules the generic SUNMATRIX module provides two utility functions `SUNMatNewEmpty` and `SUNMatVCopyOps`.

SUNMatNewEmpty

Call `A = SUNMatNewEmpty();`

Description The function `SUNMatNewEmpty` allocates a new generic `SUNMATRIX` object and initializes its content pointer and the function pointers in the operations structure to `NULL`.

Arguments None

Return value This function returns a `SUNMatrix` object. If an error occurs when allocating the object, then this routine will return `NULL`.

F2003 Name `FSUNMatNewEmpty`

SUNMatFreeEmpty

Call `SUNMatFreeEmpty(A);`

Description This routine frees the generic `SUNMatrix` object, under the assumption that any implementation-specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the ops pointer is `NULL`, and, if it is not, it will free it as well.

Arguments `A (SUNMatrix)` a `SUNMatrix` object

Return value None

F2003 Name `FSUNMatFreeEmpty`

SUNMatCopyOps

Call `retval = SUNMatCopyOps(A, B);`

Description The function `SUNMatCopyOps` copies the function pointers in the `ops` structure of `A` into the `ops` structure of `B`.

Arguments `A (SUNMatrix)` the matrix to copy operations from
`B (SUNMatrix)` the matrix to copy operations to

Return value This returns 0 if successful and a non-zero value if either of the inputs are `NULL` or the `ops` structure of either input is `NULL`.

F2003 Name `FSUNMatCopyOps`

8.1.3 SUNMatrix return codes

The functions provided to `SUNMATRIX` modules within the `SUNDIALS`-provided `SUNMATRIX` implementations utilize a common set of return codes, shown in Table 8.1. These adhere to a common pattern: 0 indicates success, and a negative value indicates a failure. The actual values of each return code are primarily to provide additional information to the user in case of a failure.

Table 8.1: Description of the `SUNMatrix` return codes

Name	Value	Description
<code>SUNMAT_SUCCESS</code>	0	successful call or converged solve
<i>continued on next page</i>		

Table 8.2: Identifiers associated with matrix kernels supplied with SUNDIALS.

Matrix ID	Matrix type	ID Value
SUNMATRIX_DENSE	Dense $M \times N$ matrix	0
SUNMATRIX_BAND	Band $M \times M$ matrix	1
SUNMATRIX_MAGMADENSE	Magma dense $M \times N$ matrix	2
SUNMATRIX_SPARSE	Sparse (CSR or CSC) $M \times N$ matrix	3
SUNMATRIX_SLUNRLOC	Adapter for the SuperLU_DIST SuperMatrix	4
SUNMATRIX_CUSPARSE	CUDA sparse CSR matrix	5
SUNMATRIX_CUSTOM	User-provided custom matrix	6

Name	Value	Description
SUNMAT_ILL_INPUT	-701	an illegal input has been provided to the function
SUNMAT_MEM_FAIL	-702	failed memory access or allocation
SUNMAT_OPERATION_FAIL	-703	a SUNMatrix operation returned nonzero
SUNMAT_MATVEC_SETUP_REQUIRED	-704	the SUNMatMatvecSetup routine needs to be called before calling SUNMatMatvec

8.1.4 SUNMatrix identifiers

Each SUNMATRIX implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 8.2. It is recommended that a user-supplied SUNMATRIX implementation use the SUNMATRIX_CUSTOM identifier.

8.1.5 Compatibility of SUNMatrix modules

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 9. More specifically, in Table 8.3 we show the matrix interfaces available as SUNMATRIX modules, and the compatible vector implementations.

Table 8.3: SUNDIALS matrix interfaces and vector implementations that can be used for each.

Matrix Interface	Serial	Parallel (MPI)	OpenMP	pThreads	hypr Vec.	PETSc Vec.	CUDA	RAJA	User Suppl.
Dense	✓		✓	✓					✓
Band	✓		✓	✓					✓
Sparse	✓		✓	✓					✓
SLUNRloc	✓	✓	✓	✓	✓	✓			✓
User supplied	✓	✓	✓	✓	✓	✓	✓	✓	✓

8.1.6 The generic SUNMatrix module implementation

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 (*matvecsetup)(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 illustrate this point we show below the implementation of a typical matrix operation from the generic SUNMATRIX module, namely `SUNMatZero`, which sets all values of a matrix `A` to zero, returning a flag denoting a successful/failed operation:

```
int SUNMatZero(SUNMatrix A)
{
    return((int) A->ops->zero(A));
}
```

Section 8.1.1 contains a complete list of all matrix operations defined by the generic SUNMATRIX module.

The Fortran 2003 interface provides a `bind(C)` derived-type for the `_generic_SUNMatrix` and the `_generic_SUNMatrix_Ops` structures. Their definition is given below.

```
type, bind(C), public :: SUNMatrix
    type(C_PTR), public :: content
    type(C_PTR), public :: ops
end type SUNMatrix

type, bind(C), public :: SUNMatrix_Ops
    type(C_FUNPTR), public :: getid
    type(C_FUNPTR), public :: clone
    type(C_FUNPTR), public :: destroy
    type(C_FUNPTR), public :: zero
    type(C_FUNPTR), public :: copy
    type(C_FUNPTR), public :: scaleadd
    type(C_FUNPTR), public :: scaleaddi
    type(C_FUNPTR), public :: matvecsetup
    type(C_FUNPTR), public :: matvec
    type(C_FUNPTR), public :: space
end type SUNMatrix_Ops
```

8.1.7 Implementing a custom SUNMatrix

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`.

It is recommended that a user-supplied SUNMATRIX implementation use the `SUNMATRIX_CUSTOM` identifier.

To aid in the creation of custom SUNMATRIX modules the generic SUNMATRIX module provides two utility functions `SUNMatNewEmpty` and `SUNMatVCopyOps`. When used in custom SUNMATRIX constructors and clone routines these functions will ease the introduction of any new optional matrix operations to the SUNMATRIX API by ensuring only required operations need to be set and all operations are copied when cloning a matrix. These functions are described in Section 8.1.2.

8.2 SUNMatrix functions used by KINSOL

In Table 8.4 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 8.4: List of matrix functions usage by KINSOL code modules

	KINLS	KINBBDPRE
<code>SUNMatGetID</code>	✓	
<code>SUNMatDestroy</code>		✓
<code>SUNMatZero</code>	✓	✓
<code>SUNMatSpace</code>		†

The matrix functions listed in Section 8.1.1 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 Section 8.1.1 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.

8.3 The SUNMatrix_Dense implementation

The dense implementation of the SUNMATRIX module provided with SUNDIALS, `SUNMATRIX_DENSE`, defines the *content* field of `SUNMatrix` to be the following structure:

```
struct _SUNMatrixContent_Dense {
    sunindextype M;
    sunindextype N;
    realtype *data;
    sunindextype ldata;
    realtype **cols;
};
```

These entries of the *content* field contain the following information:

`M` - number of rows

`N` - number of columns

`data` - pointer to a contiguous block of `realtype` variables. The elements of the dense matrix are stored columnwise, i.e. the (i,j) -th element of a dense SUNMATRIX `A` (with $0 \leq i < M$ and $0 \leq j < N$) may be accessed via `data[j*M+i]`.

`ldata` - length of the data array (= $M \cdot N$).

`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 $0 \leq i < M$ 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.

8.3.1 SUNMatrix_Dense accessor macros

The following macros are provided to access the content of a `SUNMATRIX_DENSE` matrix. The prefix `SM_` in the names denotes that these macros are for *SUNMatrix* implementations, and the suffix `_D` denotes that these are specific to the *dense* version.

- `SM_CONTENT_D`

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

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

Implementation:

```
#define SM_CONTENT_D(A)      ( (SUNMatrixContent_Dense)(A->content) )
```

- `SM_ROWS_D`, `SM_COLUMNS_D`, and `SM_LDATA_D`

These macros give individual access to various lengths relevant to the content of a dense `SUNMatrix`.

These may be used either to retrieve or to set these values. For example, the assignment `A_rows = SM_ROWS_D(A)` sets `A_rows` to be the number of rows in the matrix `A`. Similarly, the assignment `SM_COLUMNS_D(A) = A_cols` sets the number of columns in `A` to equal `A_cols`.

Implementation:

```
#define SM_ROWS_D(A)      ( SM_CONTENT_D(A)->M )
#define SM_COLUMNS_D(A)   ( SM_CONTENT_D(A)->N )
#define SM_LDATA_D(A)     ( SM_CONTENT_D(A)->ldata )
```

- SM_DATA_D and SM_COLS_D

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

The assignment `A_data = SM_DATA_D(A)` sets `A_data` to be a pointer to the first component of the data array for the dense SUNMatrix `A`. The assignment `SM_DATA_D(A) = A_data` sets the data array of `A` to be `A_data` by storing the pointer `A_data`.

Similarly, the assignment `A_cols = SM_COLS_D(A)` sets `A_cols` to be a pointer to the array of column pointers for the dense SUNMatrix `A`. The assignment `SM_COLS_D(A) = A_cols` sets the column pointer array of `A` to be `A_cols` by storing the pointer `A_cols`.

Implementation:

```
#define SM_DATA_D(A)      ( SM_CONTENT_D(A)->data )
#define SM_COLS_D(A)     ( SM_CONTENT_D(A)->cols )
```

- SM_COLUMN_D and SM_ELEMENT_D

These macros give access to the individual columns and entries of the data array of a dense SUNMatrix.

The assignment `col_j = SM_COLUMN_D(A,j)` sets `col_j` to be a pointer to the first entry of the j -th column of the $M \times N$ dense matrix `A` (with $0 \leq j < N$). The type of the expression `SM_COLUMN_D(A,j)` is `realtype *`. The pointer returned by the call `SM_COLUMN_D(A,j)` can be treated as an array which is indexed from 0 to $M - 1$.

The assignments `SM_ELEMENT_D(A,i,j) = a_ij` and `a_ij = SM_ELEMENT_D(A,i,j)` reference the (i,j) -th element of the $M \times N$ dense matrix `A` (with $0 \leq i < M$ and $0 \leq j < N$).

Implementation:

```
#define SM_COLUMN_D(A,j)  ( (SM_CONTENT_D(A)->cols)[j] )
#define SM_ELEMENT_D(A,i,j) ( (SM_CONTENT_D(A)->cols)[j][i] )
```

8.3.2 SUNMatrix_Dense functions

The `SUNMATRIX_DENSE` module defines dense implementations of all matrix operations listed in Section 8.1.1. Their names are obtained from those in Section 8.1.1 by appending the suffix `_Dense` (e.g. `SUNMatCopy_Dense`). All the standard matrix operations listed in Section 8.1.1 with the suffix `_Dense` appended are callable via the FORTRAN 2003 interface by prepending an ‘F’ (e.g. `FSUNMatCopy_Dense`).

The module `SUNMATRIX_DENSE` provides the following additional user-callable routines:

SUNDenseMatrix

Prototype `SUNMatrix SUNDenseMatrix(sunindextype M, sunindextype N)`

Description 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.

F2003 Name This function is callable as `FSUNDenseMatrix` when using the Fortran 2003 interface module.

SUNDenseMatrix_Print

Prototype `void SUNDenseMatrix_Print(SUNMatrix A, FILE* outfile)`

Description 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.

SUNDenseMatrix_Rows

Prototype `sunindextype SUNDenseMatrix_Rows(SUNMatrix A)`

Description This function returns the number of rows in the dense `SUNMatrix`.

F2003 Name This function is callable as `FSUNDenseMatrix_Rows` when using the Fortran 2003 interface module.

SUNDenseMatrix_Columns

Prototype `sunindextype SUNDenseMatrix_Columns(SUNMatrix A)`

Description This function returns the number of columns in the dense `SUNMatrix`.

F2003 Name This function is callable as `FSUNDenseMatrix_Columns` when using the Fortran 2003 interface module.

SUNDenseMatrix_LData

Prototype `sunindextype SUNDenseMatrix_LData(SUNMatrix A)`

Description This function returns the length of the data array for the dense `SUNMatrix`.

F2003 Name This function is callable as `FSUNDenseMatrix_LData` when using the Fortran 2003 interface module.

SUNDenseMatrix_Data

Prototype `realtype* SUNDenseMatrix_Data(SUNMatrix A)`

Description This function returns a pointer to the data array for the dense `SUNMatrix`.

F2003 Name This function is callable as `FSUNDenseMatrix_Data` when using the Fortran 2003 interface module.

SUNDenseMatrix_Cols

Prototype `realtype** SUNDenseMatrix_Cols(SUNMatrix A)`

Description This function returns a pointer to the `cols` array for the dense `SUNMatrix`.

SUNDenseMatrix_Column

Prototype `realtype* SUNDenseMatrix_Column(SUNMatrix A, sunindextype j)`

Description This function returns a pointer to the first entry of the j th column of the dense `SUNMatrix`. The resulting pointer should be indexed over the range 0 to $M - 1$.

F2003 Name This function is callable as `FSUNDenseMatrix_Column` when using the Fortran 2003 interface module.

Notes

- When looping over the components of a dense SUNMatrix A, the most efficient approaches are to:
 - First obtain the component array via `A_data = SM_DATA_D(A)` or `A_data = SUNDenseMatrix_Data(A)` and then access `A_data[i]` within the loop.
 - First obtain the array of column pointers via `A_cols = SM_COLS_D(A)` or `A_cols = SUNDenseMatrix_Cols(A)`, and then access `A_cols[j][i]` within the loop.
 - Within a loop over the columns, access the column pointer via `A_colj = SUNDenseMatrix_Column(A, j)` and then to access the entries within that column using `A_colj[i]` within the loop.

All three of these are more efficient than using `SM_ELEMENT_D(A, i, j)` within a double loop.

- Within the `SUNMatMatvec_Dense` routine, internal consistency checks are performed to ensure that the matrix is called with consistent NVECTOR implementations. These are currently limited to: `NVECTOR_SERIAL`, `NVECTOR_OPENMP`, and `NVECTOR_PTHREADS`. As additional compatible vector implementations are added to SUNDIALS, these will be included within this compatibility check.



8.3.3 SUNMatrix_Dense Fortran interfaces

The `SUNMATRIX_DENSE` module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The `fsummatrix_dense_mod` FORTRAN module defines interfaces to most `SUNMATRIX_DENSE` C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function `SUNDenseMatrix` is interfaced as `FSUNDenseMatrix`.

The FORTRAN 2003 `SUNMATRIX_DENSE` interface module can be accessed with the `use` statement, i.e. `use fsummatrix_dense_mod`, and linking to the library `libsundials_fsummatrixdense_mod.lib` in addition to the C library. For details on where the library and module file `fsummatrix_dense_mod.mod` are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators *without* separately linking to the `libsundials_fsummatrixdense_mod` library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN interface module, the `SUNMATRIX_DENSE` module also includes the FORTRAN-callable function `FSUNDenseMatInit(code, M, N, ier)` to initialize this `SUNMATRIX_DENSE` module for a given SUNDIALS solver. Here `code` is an integer input solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, 4 for `ARKODE`); `M` and `N` are the corresponding dense matrix construction arguments (declared to match C type `long int`); and `ier` is an error return flag equal to 0 for success and -1 for failure. Both `code` and `ier` are declared to match C type `int`. Additionally, when using `ARKODE` with a non-identity mass matrix, the FORTRAN-callable function `FSUNDenseMassMatInit(M, N, ier)` initializes this `SUNMATRIX_DENSE` module for storing the mass matrix.

8.4 The SUNMatrix_Band implementation

The banded implementation of the `SUNMATRIX` module provided with SUNDIALS, `SUNMATRIX_BAND`, defines the `content` field of `SUNMatrix` to be the following structure:

```

struct _SUNMatrixContent_Band {
    sunindextype M;
    sunindextype N;
    sunindextype mu;
    sunindextype ml;
    sunindextype s_mu;
    sunindextype ldim;
    realtype *data;
    sunindextype ldata;
    realtype **cols;
};

```

A diagram of the underlying data representation in a banded matrix is shown in Figure 8.1. 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, $0 \leq \text{mu} < N$

ml - lower half-bandwidth, $0 \leq \text{ml} < N$

s_mu - storage upper bandwidth, $\text{mu} \leq \text{s_mu} < N$. 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 $\min(N-1, \text{mu}+\text{ml})$ because of partial pivoting. The **s_mu** field holds the upper half-bandwidth allocated for A.

ldim - leading dimension ($\text{ldim} \geq \text{s_mu}+\text{ml}+1$)

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 ($= \text{ldim} \cdot N$)

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 $j-\text{mu} \leq i \leq j+\text{ml}$.

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.

8.4.1 SUNMatrix_Band accessor macros

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.

- **SM_CONTENT_B**

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

The assignment `A_cont = SM_CONTENT_B(A)` sets **A_cont** to be a pointer to the banded *SUNMatrix* content structure.

Implementation:

```
#define SM_CONTENT_B(A)      ( (SUNMatrixContent_Band)(A->content) )
```

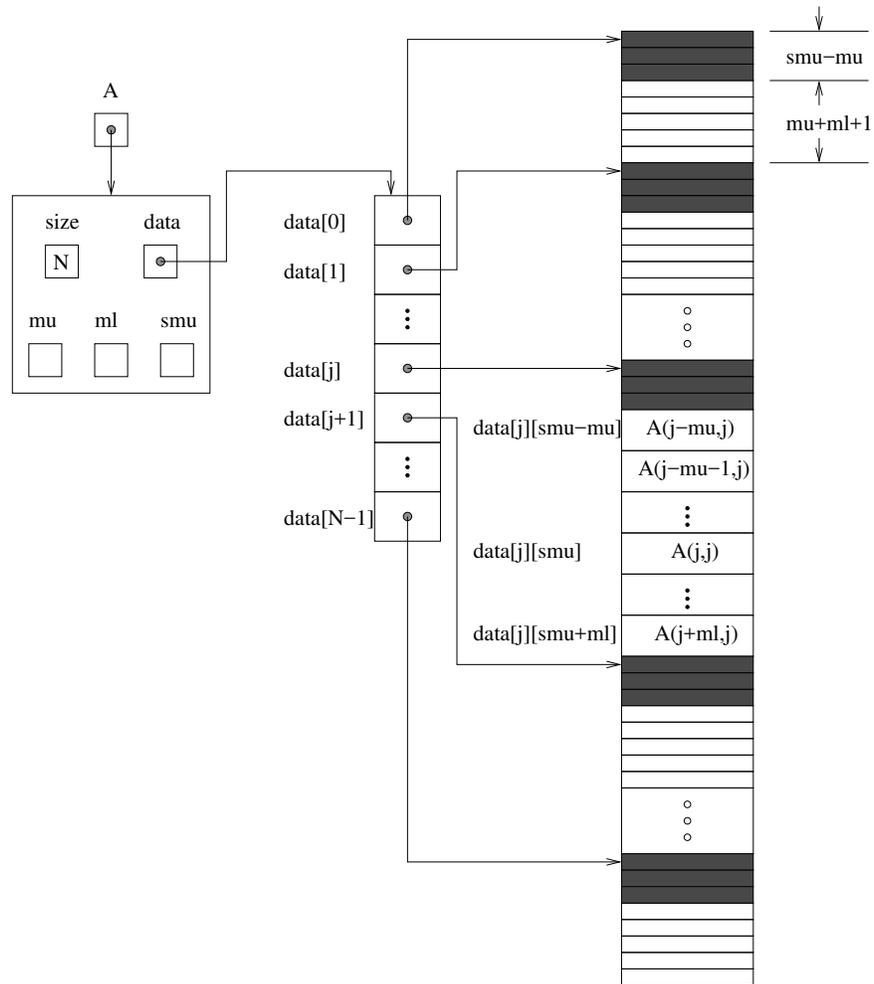


Figure 8.1: Diagram of the storage for the SUNMATRIX_BAND module. Here A is an $N \times N$ band matrix with upper and lower half-bandwidths μ and m_1 , respectively. The rows and columns of A are numbered from 0 to $N - 1$ and the (i, j) -th element of A is denoted $A(i, j)$. The greyed out areas of the underlying component storage are used by the associated SUNLINSOL_BAND linear solver.

- `SM_ROWS_B`, `SM_COLUMNS_B`, `SM_UBAND_B`, `SM_LBAND_B`, `SM_SUBAND_B`, `SM_LDIM_B`, and `SM_LDATA_B`

These macros give individual access to various lengths relevant to the content of a banded SUNMatrix.

These may be used either to retrieve or to set these values. For example, the assignment `A_rows = SM_ROWS_B(A)` sets `A_rows` to be the number of rows in the matrix `A`. Similarly, the assignment `SM_COLUMNS_B(A) = A_cols` sets the number of columns in `A` to equal `A_cols`.

Implementation:

```
#define SM_ROWS_B(A)      ( SM_CONTENT_B(A)->M )
#define SM_COLUMNS_B(A)   ( SM_CONTENT_B(A)->N )
#define SM_UBAND_B(A)    ( SM_CONTENT_B(A)->mu )
#define SM_LBAND_B(A)    ( SM_CONTENT_B(A)->m1 )
#define SM_SUBAND_B(A)   ( SM_CONTENT_B(A)->s_mu )
#define SM_LDIM_B(A)     ( SM_CONTENT_B(A)->ldim )
#define SM_LDATA_B(A)    ( SM_CONTENT_B(A)->ldata )
```

- `SM_DATA_B` and `SM_COLS_B`

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

The assignment `A_data = SM_DATA_B(A)` sets `A_data` to be a pointer to the first component of the data array for the banded SUNMatrix `A`. The assignment `SM_DATA_B(A) = A_data` sets the data array of `A` to be `A_data` by storing the pointer `A_data`.

Similarly, the assignment `A_cols = SM_COLS_B(A)` sets `A_cols` to be a pointer to the array of column pointers for the banded SUNMatrix `A`. The assignment `SM_COLS_B(A) = A_cols` sets the column pointer array of `A` to be `A_cols` by storing the pointer `A_cols`.

Implementation:

```
#define SM_DATA_B(A)      ( SM_CONTENT_B(A)->data )
#define SM_COLS_B(A)     ( SM_CONTENT_B(A)->cols )
```

- `SM_COLUMN_B`, `SM_COLUMN_ELEMENT_B`, and `SM_ELEMENT_B`

These macros give access to the individual columns and entries of the data array of a banded SUNMatrix.

The assignments `SM_ELEMENT_B(A,i,j) = a_ij` and `a_ij = SM_ELEMENT_B(A,i,j)` reference the (i,j) -th element of the $N \times N$ band matrix `A`, where $0 \leq i, j \leq N - 1$. The location (i,j) should further satisfy $j - \mu \leq i \leq j + m1$.

The assignment `col_j = SM_COLUMN_B(A,j)` sets `col_j` to be a pointer to the diagonal element of the j -th column of the $N \times N$ band matrix `A`, $0 \leq j \leq N - 1$. The type of the expression `SM_COLUMN_B(A,j)` is `realtype *`. The pointer returned by the call `SM_COLUMN_B(A,j)` can be treated as an array which is indexed from $-\mu$ to $m1$.

The assignments `SM_COLUMN_ELEMENT_B(col_j,i,j) = a_ij` and `a_ij = SM_COLUMN_ELEMENT_B(col_j,i,j)` reference the (i,j) -th entry of the band matrix `A` when used in conjunction with `SM_COLUMN_B` to reference the j -th column through `col_j`. The index (i,j) should satisfy $j - \mu \leq i \leq j + m1$.

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)] )
```

8.4.2 SUNMatrix_Band functions

The SUNMATRIX_BAND module defines banded implementations of all matrix operations listed in Section 8.1.1. Their names are obtained from those in Section 8.1.1 by appending the suffix `_Band` (e.g. `SUNMatCopy_Band`). All the standard matrix operations listed in Section 8.1.1 with the suffix `_Band` appended are callable via the FORTRAN 2003 interface by prepending an ‘F’ (e.g. `FSUNMatCopy_Band`).

The module SUNMATRIX_BAND provides the following additional user-callable routines:

SUNBandMatrix

Prototype `SUNMatrix SUNBandMatrix(sunindextype N, sunindextype mu, sunindextype ml)`

Description This constructor function creates and allocates memory for a banded `SUNMatrix`. Its arguments are the matrix size, `N`, and the upper and lower half-bandwidths of the matrix, `mu` and `ml`. The stored upper bandwidth is set to `mu+ml` to accommodate subsequent factorization in the `SUNLINSOL_BAND` and `SUNLINSOL_LAPACKBAND` modules.

F2003 Name This function is callable as `FSUNBandMatrix` when using the Fortran 2003 interface module.

SUNBandMatrixStorage

Prototype `SUNMatrix SUNBandMatrixStorage(sunindextype N, sunindextype mu, sunindextype ml, sunindextype smu)`

Description 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.

Note: it is strongly recommended that users call the default constructor, `SUNBandMatrix`, in all standard use cases. This advanced constructor is used internally within `SUNDIALS` solvers, and is provided to users who require banded matrices for non-default purposes.

SUNBandMatrix_Print

Prototype `void SUNBandMatrix_Print(SUNMatrix A, FILE* outfile)`

Description 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.

SUNBandMatrix_Rows

Prototype `sunindextype SUNBandMatrix_Rows(SUNMatrix A)`

Description This function returns the number of rows in the banded `SUNMatrix`.

F2003 Name This function is callable as `FSUNBandMatrix_Rows` when using the Fortran 2003 interface module.

SUNBandMatrix_Columns

Prototype `sunindextype SUNBandMatrix_Columns(SUNMatrix A)`

Description This function returns the number of columns in the banded `SUNMatrix`.

F2003 Name This function is callable as `FSUNBandMatrix_Columns` when using the Fortran 2003 interface module.

SUNBandMatrix_LowerBandwidth

Prototype `sunindextype SUNBandMatrix_LowerBandwidth(SUNMatrix A)`

Description This function returns the lower half-bandwidth of the banded `SUNMatrix`.

F2003 Name This function is callable as `FSUNBandMatrix_LowerBandwidth` when using the Fortran 2003 interface module.

SUNBandMatrix_UpperBandwidth

Prototype `sunindextype SUNBandMatrix_UpperBandwidth(SUNMatrix A)`

Description This function returns the upper half-bandwidth of the banded `SUNMatrix`.

F2003 Name This function is callable as `FSUNBandMatrix_UpperBandwidth` when using the Fortran 2003 interface module.

SUNBandMatrix_StoredUpperBandwidth

Prototype `sunindextype SUNBandMatrix_StoredUpperBandwidth(SUNMatrix A)`

Description This function returns the stored upper half-bandwidth of the banded `SUNMatrix`.

F2003 Name This function is callable as `FSUNBandMatrix_StoredUpperBandwidth` when using the Fortran 2003 interface module.

SUNBandMatrix_LDim

Prototype `sunindextype SUNBandMatrix_LDim(SUNMatrix A)`

Description This function returns the length of the leading dimension of the banded `SUNMatrix`.

F2003 Name This function is callable as `FSUNBandMatrix_LDim` when using the Fortran 2003 interface module.

SUNBandMatrix_Data

Prototype `realtype* SUNBandMatrix_Data(SUNMatrix A)`

Description This function returns a pointer to the data array for the banded `SUNMatrix`.

F2003 Name This function is callable as `FSUNBandMatrix_Data` when using the Fortran 2003 interface module.

SUNBandMatrix_Cols

Prototype `realtype** SUNBandMatrix_Cols(SUNMatrix A)`

Description This function returns a pointer to the `cols` array for the banded `SUNMatrix`.

SUNBandMatrix_Column

Prototype `realtype* SUNBandMatrix_Column(SUNMatrix A, sunindextype j)`

Description 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 $m1$.

F2003 Name This function is callable as `FSUNBandMatrix_Column` when using the Fortran 2003 interface module.

Notes

- When looping over the components of a banded SUNMatrix A, the most efficient approaches are to:
 - First obtain the component array via `A_data = SM_DATA_B(A)` or `A_data = SUNBandMatrix_Data(A)` and then access `A_data[i]` within the loop.
 - First obtain the array of column pointers via `A_cols = SM_COLS_B(A)` or `A_cols = SUNBandMatrix_Cols(A)`, and then access `A_cols[j][i]` within the loop.
 - Within a loop over the columns, access the column pointer via `A_colj = SUNBandMatrix_Column(A, j)` and then to access the entries within that column using `SM_COLUMN_ELEMENT_B(A_colj, i, j)`.

All three of these are more efficient than using `SM_ELEMENT_B(A, i, j)` within a double loop.

- Within the `SUNMatMatvec_Band` routine, internal consistency checks are performed to ensure that the matrix is called with consistent NVECTOR implementations. These are currently limited to: `NVECTOR_SERIAL`, `NVECTOR_OPENMP`, and `NVECTOR_PTHREADS`. As additional compatible vector implementations are added to SUNDIALS, these will be included within this compatibility check.

**8.4.3 SUNMatrix_Band Fortran interfaces**

The `SUNMATRIX_BAND` module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The `f_sunmatrix_band_mod` FORTRAN module defines interfaces to most `SUNMATRIX_BAND` C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function `SUNBandMatrix` is interfaced as `FSUNBandMatrix`.

The FORTRAN 2003 `SUNMATRIX_BAND` interface module can be accessed with the `use` statement, i.e. `use f_sunmatrix_band_mod`, and linking to the library `libsundials_f_sunmatrixband_mod.lib` in addition to the C library. For details on where the library and module file `f_sunmatrix_band_mod.mod` are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators *without* separately linking to the `libsundials_f_sunmatrixband_mod` library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN interface module, the `SUNMATRIX_BAND` module also includes the FORTRAN-callable function `FSUNBandMatInit(code, N, mu, m1, ier)` to initialize this `SUNMATRIX_BAND` module for a given SUNDIALS solver. Here `code` is an integer input solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, 4 for `ARKODE`); `N`, `mu`, and `m1` 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, m1, ier)` initializes this `SUNMATRIX_BAND` module for storing the mass matrix.

8.5 The SUNMatrix_Sparse implementation

The sparse implementation of the SUNMATRIX module provided with SUNDIALS, SUNMATRIX_SPARSE, is designed to work with either *compressed-sparse-column* (CSC) or *compressed-sparse-row* (CSR) sparse matrix formats. To this end, it defines the *content* field of SUNMatrix to be the following structure:

```
struct _SUNMatrixContent_Sparse {
    sunindextype M;
    sunindextype N;
    sunindextype NNZ;
    sunindextype NP;
    realtype *data;
    int sparsetype;
    sunindextype *indexvals;
    sunindextype *indexptrs;
    /* CSC indices */
    sunindextype **rowvals;
    sunindextype **colptrs;
    /* CSR indices */
    sunindextype **colvals;
    sunindextype **rowptrs;
};
```

A diagram of the underlying data representation for a CSC matrix is shown in Figure 8.2 (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 5×4 CSC matrix

$$\begin{bmatrix} 0 & 3 & 1 & 0 \\ 3 & 0 & 0 & 2 \\ 0 & 7 & 0 & 0 \\ 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 5 \end{bmatrix}$$

could be stored in this structure as either

```
M = 5;
N = 4;
NNZ = 8;
NP = N;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0};
sparsetype = CSC_MAT;
indexvals = {1, 3, 0, 2, 0, 1, 3, 4};
indexptrs = {0, 2, 4, 5, 8};
```

or

```
M = 5;
N = 4;
NNZ = 10;
NP = N;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0, *, *};
sparsetype = CSC_MAT;
indexvals = {1, 3, 0, 2, 0, 1, 3, 4, *, *};
indexptrs = {0, 2, 4, 5, 8};
```

where the first has no unused space, and the second has additional storage (the entries marked with * may contain any values). Note in both cases that the final value in `indexptrs` is 8, indicating the total number of nonzero entries in the matrix.

Similarly, in CSR format, the same matrix could be stored as

```
M = 5;
N = 4;
NNZ = 8;
NP = M;
data = {3.0, 1.0, 3.0, 2.0, 7.0, 1.0, 9.0, 5.0};
sparsetype = CSR_MAT;
indexvals = {1, 2, 0, 3, 1, 0, 3, 3};
indexptrs = {0, 2, 4, 5, 7, 8};
```

The header file to include when using this module is `sunmatrix/sunmatrix_sparse.h`. The `SUNMATRIX_SPARSE` module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunmatrixsparse` module library.

8.5.1 SUNMatrix_Sparse accessor macros

The following macros are provided to access the content of a `SUNMATRIX_SPARSE` matrix. The prefix `SM_` in the names denotes that these macros are for *SUNMatrix* implementations, and the suffix `_S` denotes that these are specific to the *sparse* version.

- `SM_CONTENT_S`

This routine gives access to the contents of the sparse `SUNMatrix`.

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

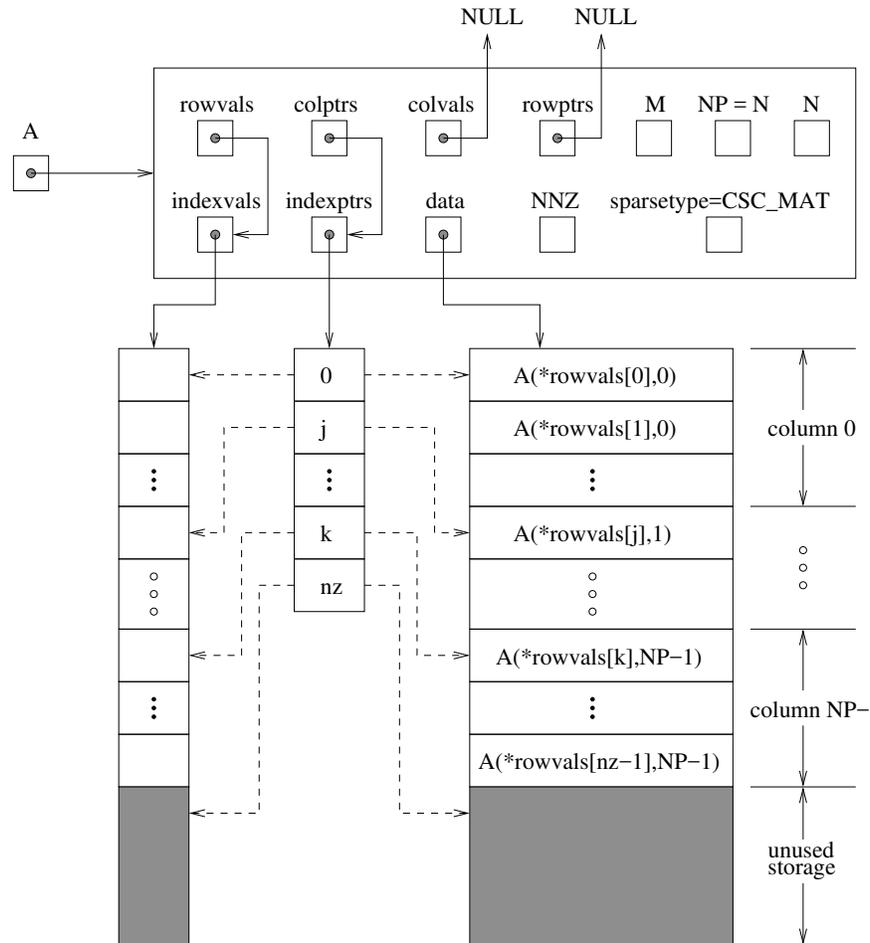


Figure 8.2: Diagram of the storage for a compressed-sparse-column matrix. Here A is an $M \times N$ sparse matrix with storage for up to NNZ nonzero entries (the allocated length of both `data` and `indexvals`). The entries in `indexvals` may assume values from 0 to $M - 1$, corresponding to the row 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.

Implementation:

```
#define SM_CONTENT_S(A)      ( (SUNMatrixContent_Sparse)(A->content) )
```

- SM_ROWS_S, SM_COLUMNS_S, SM_NNZ_S, SM_NP_S, and SM_SPARSETYPE_S

These macros give individual access to various lengths relevant to the content of a sparse SUNMatrix.

These may be used either to retrieve or to set these values. For example, the assignment `A_rows = SM_ROWS_S(A)` sets `A_rows` to be the number of rows in the matrix `A`. Similarly, the assignment `SM_COLUMNS_S(A) = A_cols` sets the number of columns in `A` to equal `A_cols`.

Implementation:

```
#define SM_ROWS_S(A)          ( SM_CONTENT_S(A)->M )
#define SM_COLUMNS_S(A)       ( SM_CONTENT_S(A)->N )
#define SM_NNZ_S(A)           ( SM_CONTENT_S(A)->NNZ )
#define SM_NP_S(A)            ( SM_CONTENT_S(A)->NP )
#define SM_SPARSETYPE_S(A)    ( SM_CONTENT_S(A)->sparsetype )
```

- SM_DATA_S, SM_INDEXVALS_S, and SM_INDEXPTRS_S

These macros give access to the `data` and `index` arrays for the matrix entries.

The assignment `A_data = SM_DATA_S(A)` sets `A_data` to be a pointer to the first component of the data array for the sparse SUNMatrix `A`. The assignment `SM_DATA_S(A) = A_data` sets the data array of `A` to be `A_data` by storing the pointer `A_data`.

Similarly, the assignment `A_indexvals = SM_INDEXVALS_S(A)` sets `A_indexvals` to be a pointer to the array of index values (i.e. row indices for a CSC matrix, or column indices for a CSR matrix) for the sparse SUNMatrix `A`. The assignment `A_indexptrs = SM_INDEXPTRS_S(A)` sets `A_indexptrs` to be a pointer to the array of index pointers (i.e. the starting indices in the data/indexvals arrays for each row or column in CSR or CSC formats, respectively).

Implementation:

```
#define SM_DATA_S(A)          ( SM_CONTENT_S(A)->data )
#define SM_INDEXVALS_S(A)     ( SM_CONTENT_S(A)->indexvals )
#define SM_INDEXPTRS_S(A)     ( SM_CONTENT_S(A)->indexptrs )
```

8.5.2 SUNMatrix_Sparse functions

The `SUNMATRIX_SPARSE` module defines sparse implementations of all matrix operations listed in Section 8.1.1. Their names are obtained from those in Section 8.1.1 by appending the suffix `_Sparse` (e.g. `SUNMatCopy_Sparse`). All the standard matrix operations listed in Section 8.1.1 with the suffix `_Sparse` appended are callable via the FORTRAN 2003 interface by prepending an ‘F’ (e.g. `FSUNMatCopy_Sparse`).

The module `SUNMATRIX_SPARSE` provides the following additional user-callable routines:

SUNSparseMatrix

Prototype `SUNMatrix SUNSparseMatrix(sunindextype M, sunindextype N, sunindextype NNZ, int sparsetype)`

Description This function creates and allocates memory for a sparse SUNMatrix. Its arguments are the number of rows and columns of the matrix, `M` and `N`, the maximum number of nonzeros to be stored in the matrix, `NNZ`, and a flag `sparsetype` indicating whether to use CSR or CSC format (valid arguments are `CSR_MAT` or `CSC_MAT`).

F2003 Name This function is callable as `FSUNSparseMatrix` when using the Fortran 2003 interface module.

SUNSparseFromDenseMatrix

Prototype `SUNMatrix SUNSparseFromDenseMatrix(SUNMatrix A, realtype droptol, int sparsetype);`

Description This function creates a new sparse matrix from an existing dense matrix by copying all values with magnitude larger than `droptol` into the sparse matrix structure.

Requirements:

- `A` must have type `SUNMATRIX_DENSE`;
- `droptol` must be non-negative;
- `sparsetype` must be either `CSC_MAT` or `CSR_MAT`.

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

F2003 Name This function is callable as `FSUNSparseFromDenseMatrix` when using the Fortran 2003 interface module.

SUNSparseFromBandMatrix

Prototype `SUNMatrix SUNSparseFromBandMatrix(SUNMatrix A, realtype droptol, int sparsetype);`

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

Requirements:

- `A` must have type `SUNMATRIX_BAND`;
- `droptol` must be non-negative;
- `sparsetype` must be either `CSC_MAT` or `CSR_MAT`.

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

F2003 Name This function is callable as `FSUNSparseFromBandMatrix` when using the Fortran 2003 interface module.

SUNSparseMatrix_Realloc

Prototype `int SUNSparseMatrix_Realloc(SUNMatrix A)`

Description This function reallocates internal storage arrays in a sparse matrix so that the resulting sparse matrix has no wasted space (i.e. the space allocated for nonzero entries equals the actual number of nonzeros, `indexptrs[NP]`). Returns 0 on success and 1 on failure (e.g. if the input matrix is not sparse).

F2003 Name This function is callable as `FSUNSparseMatrix_Realloc` when using the Fortran 2003 interface module.

SUNSparseMatrix_Reallocate

Prototype `int SUNSparseMatrix_Reallocate(SUNMatrix A, sunindextype NNZ)`

Description 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 `NNZ` is negative).

F2003 Name This function is callable as `FSUNSparseMatrix_Reallocate` when using the Fortran 2003 interface module.

SUNSparseMatrix_Print

Prototype `void SUNSparseMatrix_Print(SUNMatrix A, FILE* outfile)`

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

SUNSparseMatrix_Rows

Prototype `sunindextype SUNSparseMatrix_Rows(SUNMatrix A)`

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

F2003 Name This function is callable as `FSUNSparseMatrix_Rows` when using the Fortran 2003 interface module.

SUNSparseMatrix_Columns

Prototype `sunindextype SUNSparseMatrix_Columns(SUNMatrix A)`

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

F2003 Name This function is callable as `FSUNSparseMatrix_Columns` when using the Fortran 2003 interface module.

SUNSparseMatrix_NNZ

Prototype `sunindextype SUNSparseMatrix_NNZ(SUNMatrix A)`

Description This function returns the number of entries allocated for nonzero storage for the sparse matrix `SUNMatrix`.

F2003 Name This function is callable as `FSUNSparseMatrix_NNZ` when using the Fortran 2003 interface module.

SUNSparseMatrix_NP

Prototype `sunindextype SUNSparseMatrix_NP(SUNMatrix A)`

Description 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.

F2003 Name This function is callable as `FSUNSparseMatrix_NP` when using the Fortran 2003 interface module.

SUNSparseMatrix_SparseType

Prototype `int SUNSparseMatrix_SparseType(SUNMatrix A)`

Description This function returns the storage type (`CSR_MAT` or `CSC_MAT`) for the sparse `SUNMatrix`.

F2003 Name This function is callable as `FSUNSparseMatrix_SparseType` when using the Fortran 2003 interface module.

SUNSparseMatrix_Data

Prototype `realtype* SUNSparseMatrix_Data(SUNMatrix A)`

Description This function returns a pointer to the data array for the sparse `SUNMatrix`.

F2003 Name This function is callable as `FSUNSparseMatrix_Data` when using the Fortran 2003 interface module.

SUNSparseMatrix_IndexValues

Prototype `sunindextype* SUNSparseMatrix_IndexValues(SUNMatrix A)`

Description 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.

F2003 Name This function is callable as `FSUNSparseMatrix_IndexValues` when using the Fortran 2003 interface module.

SUNSparseMatrix_IndexPointers

Prototype `sunindextype* SUNSparseMatrix_IndexPointers(SUNMatrix A)`

Description 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.

F2003 Name This function is callable as `FSUNSparseMatrix_IndexPointers` when using the Fortran 2003 interface module.



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`, `NVECTOR_PTHREADS`, and `NVECTOR_CUDA` when using managed memory. As additional compatible vector implementations are added to `SUNDIALS`, these will be included within this compatibility check.

8.5.3 SUNMatrix_Sparse Fortran interfaces

The `SUNMATRIX_SPARSE` module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The `f_sunmatrix_sparse_mod` FORTRAN module defines interfaces to most `SUNMATRIX_SPARSE` C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function `SUNSparseMatrix` is interfaced as `FSUNSparseMatrix`.

The FORTRAN 2003 `SUNMATRIX_SPARSE` interface module can be accessed with the `use` statement, i.e. `use f_sunmatrix_sparse_mod`, and linking to the library `libsundials_f_sunmatrixsparse_mod.lib` in addition to the C library. For details on where the library and module file `f_sunmatrix_sparse_mod.mod` are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 `SUNDIALS` integrators *without* separately linking to the `libsundials_f_sunmatrixsparse_mod` library.

FORTRAN 77 interface functions

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.

8.6 The SUNMatrix_SLUNRloc implementation

The SUNMATRIX_SLUNRLOC implementation of the SUNMATRIX module provided with SUNDIALS is an adapter for the SuperMatrix structure provided by the SuperLU_DIST sparse matrix factorization and solver library written by X. Sherry Li [8, 24, 32, 33]. It is designed to be used with the SUNLINSOL_SUPERLUDIST linear solver discussed in Section 9.10. To this end, it defines the *content* field of SUNMatrix to be the following structure:

```
struct _SUNMatrixContent_SLUNRloc {
    booleantype    own_data;
    gridinfo_t     *grid;
    sunindextype   *row_to_proc;
    pdgsmv_comm_t *gsmv_comm;
    SuperMatrix    *A_super;
    SuperMatrix    *ACS_super;
};
```

A more complete description of the this *content* field is given below:

own_data - a flag which indicates if the SUNMatrix is responsible for freeing **A_super**

grid - pointer to the SuperLU_DIST structure that stores the 2D process grid

row_to_proc - a mapping between the rows in the matrix and the process it resides on; will be NULL until the SUNMatMatvecSetup routine is called

gsmv_comm - pointer to the SuperLU_DIST structure that stores the communication information needed for matrix-vector multiplication; will be NULL until the SUNMatMatvecSetup routine is called

A_super - pointer to the underlying SuperLU_DIST SuperMatrix with **Stype** = SLU_NR_loc, **Dtype** = SLU_D, **Mtype** = SLU_GE; must have the full diagonal present to be used with SUNMatScaleAddI routine

ACS_super - a column-sorted version of the matrix needed to perform matrix-vector multiplication; will be NULL until the routine SUNMatMatvecSetup routine is called

The header file to include when using this module is `sunmatrix/sunmatrix_slunrloc.h`. The installed module library to link to is `libsundials_sunmatrixslunrloc.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

8.6.1 SUNMatrix_SLUNRloc functions

The module SUNMATRIX_SLUNRLOC provides the following user-callable routines:

`SUNMatrix_SLUNRloc`

Call `A = SUNMatrix_SLUNRloc(Asuper, grid);`

Description The function `SUNMatrix_SLUNRloc` creates and allocates memory for a SUNMATRIX_SLUNRLOC object.

Arguments **Asuper** (**SuperMatrix***) a fully-allocated SuperLU_DIST SuperMatrix that the SUNMatrix will wrap; must have **Stype** = SLU_NR_loc, **Dtype** = SLU_D, **Mtype** = SLU_GE to be compatible
grid (**gridinfo_t***) the initialized SuperLU_DIST 2D process grid structure

Return value a SUNMatrix object if **Asuper** is compatible else NULL

Notes

`SUNMatrix_SLUNRloc_Print`

Call `SUNMatrix_SLUNRloc_Print(A, fp);`

Description The function `SUNMatrix_SLUNRloc_Print` prints the underlying `SuperMatrix` content.

Arguments `A` (`SUNMatrix`) the matrix to print
`fp` (`FILE`) the file pointer used for printing

Return value `void`

Notes

`SUNMatrix_SLUNRloc_SuperMatrix`

Call `Asuper = SUNMatrix_SLUNRloc_SuperMatrix(A);`

Description The function `SUNMatrix_SLUNRloc_SuperMatrix` provides access to the underlying `SuperLU_DIST` `SuperMatrix` of `A`.

Arguments `A` (`SUNMatrix`) the matrix to access

Return value `SuperMatrix*`

Notes

`SUNMatrix_SLUNRloc_ProcessGrid`

Call `grid = SUNMatrix_SLUNRloc_ProcessGrid(A);`

Description The function `SUNMatrix_SLUNRloc_ProcessGrid` provides access to the `SuperLU_DIST` `gridinfo_t` structure associated with `A`.

Arguments `A` (`SUNMatrix`) the matrix to access

Return value `gridinfo_t*`

Notes

`SUNMatrix_SLUNRloc_OwnData`

Call `does_own_data = SUNMatrix_SLUNRloc_OwnData(A);`

Description The function `SUNMatrix_SLUNRloc_OwnData` returns true if the `SUNMatrix` object is responsible for freeing `A.super`, otherwise it returns false.

Arguments `A` (`SUNMatrix`) the matrix to access

Return value `booleantype`

Notes

The `SUNMATRIX_SLUNRLOC` module defines implementations of all generic `SUNMatrix` operations listed in Section 8.1.1:

- `SUNMatGetID_SLUNRloc` - returns `SUNMATRIX_SLUNRLOC`
- `SUNMatClone_SLUNRloc`
- `SUNMatDestroy_SLUNRloc`
- `SUNMatSpace_SLUNRloc` - this only returns information for the storage within the matrix interface, i.e. storage for `row_to_proc`
- `SUNMatZero_SLUNRloc`
- `SUNMatCopy_SLUNRloc`

- `SUNMatScaleAdd_SLUNRloc` - performs $A = cA + B$, but A and B must have the same sparsity pattern
- `SUNMatScaleAddI_SLUNRloc` - performs $A = cA + I$, but the diagonal of A must be present
- `SUNMatMatvecSetup_SLUNRloc` - initializes the SuperLU_DIST parallel communication structures needed to perform a matrix-vector product; only needs to be called before the first call to `SUNMatMatvec` or if the matrix changed since the last setup
- `SUNMatMatvec_SLUNRloc`

The `SUNMATRIX_SLUNRLOC` module requires that the complete diagonal, i.e. nonzeros and zeros, is present in order to use the `SUNMatScaleAddI` operation.



8.7 The SUNMatrix_cuSparse implementation

The `SUNMATRIX_CUSPARSE` implementation of the `SUNMatrix` module provided with `SUNDIALS`, is an interface to the NVIDIA cuSPARSE matrix for use on NVIDIA GPUs [7]. All data stored by this matrix implementation resides on the GPU at all times. The implementation currently supports the cuSPARSE CSR matrix format described in the cuSPARSE documentation as well as a unique low-storage format for block-diagonal matrices of the form

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 & 0 & \cdots & 0 \\ 0 & \mathbf{A}_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A}_{n-1} \end{bmatrix}$$

where all the block matrices A_j share the same sparsity pattern. We will refer to this format as BCSR (not to be confused with the canonical BSR format where each block is stored as dense). In this format, the CSR column indices and row pointers are only stored for the first block and are computed only as necessary for other blocks. This can drastically reduce the amount of storage required compared to the regular CSR format when there is a large number of blocks. This format is well-suited for, and intended to be used with the `SUNLinearSolver_cuSolverSp_batchQR` linear solver (see Section 9.12).

The header file to include when using this module is `sunmatrix/sunmatrix_cusparse.h`. The installed library to link to is `libsundials_sunmatrixcusparse.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The `SUNMatrix_cuSparse` module is experimental and subject to change.



8.7.1 SUNMatrix_cuSparse functions

The `SUNMATRIX_CUSPARSE` module defines GPU-enabled sparse implementations of all matrix operations listed in Section 8.1.1 except for the `SUNMatSpace` and `SUNMatMatvecSetup` operations:

1. `SUNMatGetID_cuSparse` – returns `SUNMATRIX_CUSPARSE`
2. `SUNMatClone_cuSparse`
3. `SUNMatDestroy_cuSparse`
4. `SUNMatZero_cuSparse`
5. `SUNMatCopy_cuSparse`
6. `SUNMatScaleAdd_cuSparse` – performs $A = cA + B$, where A and B must have the same sparsity pattern

7. `SUNMatScaleAddI_cuSparse` – performs $A = cA + I$, where the diagonal of A must be present
8. `SUNMatMatvec_cuSparse`

In addition, the `SUNMATRIX_CUSPARSE` module defines the following implementation specific functions:

`SUNMatrix_cuSparse_NewCSR`

Call `A = SUNMatrix_cuSparse_NewCSR(M, N, NNZ, cusp)`

Description This constructor function creates and allocates memory for a `SUNMATRIX_CUSPARSE` `SUNMatrix` that uses the CSR storage format.

Arguments `M` (`int`) the number of matrix rows
`N` (`int`) the number of matrix columns
`NNZ` (`int`) the number of matrix nonzeros
`cusp` (`cusparseHandle_t`) a valid `cusparseHandle_t`

Return value a `SUNMatrix` object if successful else `NULL`

`SUNMatrix_cuSparse_NewBlockCSR`

Call `A = SUNMatrix_cuSparse_NewBlockCSR(nblocks, blockrows, blockcols, blocknnz, cusp)`

Description This constructor function creates and allocates memory for a `SUNMATRIX_CUSPARSE` `SUNMatrix` that leverages the `SUNMAT_CUSPARSE_BCSR` storage format to store a block diagonal matrix where each block shares the same sparsity pattern. **The blocks must be square.**

Arguments `nblocks` (`int`) the number of matrix blocks
`blockrows` (`int`) the number of rows for a block
`blockcols` (`int`) the number of columns for a block
`blocknnz` (`int`) the number of nonzeros in a block
`cusp` a valid `cusparseHandle_t`

Return value a `SUNMatrix` object if successful else `NULL`

Notes The `SUNMAT_CUSPARSE_BCSR` format currently only supports square matrices.

`SUNMatrix_cuSparse_MakeCSR`

Call `A = SUNMatrix_cuSparse_MakeCSR(mat_descr, M, N, NNZ, rowptrs, colind, data, cusp)`

Description This constructor function creates and allocates memory for a `SUNMATRIX_CUSPARSE` `SUNMatrix` that uses the CSR storage format from the user provided pointers.

Arguments `mat_descr` a valid `cusparseMatDescr_t` object; must use `CUSPARSE_INDEX_BASE_ZERO` indexing
`M` (`int`) the number of matrix rows
`N` (`int`) the number of matrix columns
`NNZ` (`int`) the number of matrix nonzeros
`rowptrs` (`int*`) a contiguous array of the CSR row pointers
`colind` (`int*`) a contiguous array of the CSR column indices
`data` (`realtype*`) a contiguous array of the nonzero data
`cusp` (`cusparseHandle_t`) a valid `cusparseHandle_t`

Return value a `SUNMatrix` object if successful else `NULL`

SUNMatrix_cuSparse_Rows

Call `M = SUNMatrix_cuSparse_Rows(A)`

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

Arguments `A` (`SUNMatrix`)

Return value the number of rows in the sparse `SUNMatrix`

SUNMatrix_cuSparse_Columns

Call `N = SUNMatrix_cuSparse_Columns(A)`

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

Arguments `A` (`SUNMatrix`)

Return value the number of columns in the sparse `SUNMatrix`

SUNMatrix_cuSparse_NNZ

Call `nnz = SUNMatrix_cuSparse_NNZ(A)`

Description This function returns the number of nonzeros in the sparse `SUNMatrix`.

Arguments `A` (`SUNMatrix`)

Return value the number of nonzeros in the sparse `SUNMatrix`

SUNMatrix_cuSparse_SparseType

Call `type = SUNMatrix_cuSparse_SparseType(A)`

Description This function returns the sparsity format for the sparse `SUNMatrix`.

Arguments `A` (`SUNMatrix`)

Return value the `SUNMAT_CUSPARSE_CSR` or `SUNMAT_CUSPARSE_BCSR` sparsity formats

SUNMatrix_cuSparse_IndexValues

Call `colind = SUNMatrix_cuSparse_IndexValues(A)`

Description This function returns a pointer to the index value array for the sparse `SUNMatrix`.

Arguments `A` (`SUNMatrix`)

Return value for the CSR format this is an array of the column indices for each nonzero entry. For the BCSR format this is an array of the column indices for each nonzero entry in the first block only.

SUNMatrix_cuSparse_IndexPointers

Call `rowptrs = SUNMatrix_cuSparse_IndexPointers(A)`

Description This function returns a pointer to the index pointers array for the sparse `SUNMatrix`.

Arguments `A` (`SUNMatrix`)

Return value for the CSR format this is an array of the locations of the first entry of each row in the `data` and `indexvalues` arrays, for the BCSR format this is an array of the locations of each row in the `data` and `indexvalues` arrays in the first block only.

SUNMatrix_cuSparse_NumBlocks

Call `nblocks = SUNMatrix_cuSparse_NumBlocks(A)`

Description This function returns the number of blocks in the sparse `SUNMatrix`.

Arguments `A` (`SUNMatrix`)

Return value the number of matrix blocks

SUNMatrix_cuSparse_BlockRows

Call `blockrows = SUNMatrix_cuSparse_BlockRows(A)`

Description This function returns the number of rows of a block of the sparse `SUNMatrix`.

Arguments `A` (`SUNMatrix`)

Return value the number of rows of a block

SUNMatrix_cuSparse_BlockColumns

Call `blockcols = SUNMatrix_cuSparse_BlockColumns(A)`

Description This function returns the number of columns of a block of the sparse `SUNMatrix`.

Arguments `A` (`SUNMatrix`)

Return value the number of columns of a block

SUNMatrix_cuSparse_BlockNNZ

Call `blockdim = SUNMatrix_cuSparse_BlockNNZ(A)`

Description This function returns the nonzeros of a block of the sparse `SUNMatrix`.

Arguments `A` (`SUNMatrix`)

Return value the number of nonzeros of a block

SUNMatrix_cuSparse_BlockData

Call `nzdata = SUNMatrix_cuSparse_BlockData(A, blockidx)`

Description This function returns a pointer to the start of the nonzero values in the data array for given block index. The first block in the `SUNMatrix` is index 0, the second block is index 1, and so on.

Arguments `A` (`SUNMatrix`)

`blockidx` (`int`) the index of the desired block

Return value a pointer to the start of the nonzero values in the data array for given block index

SUNMatrix_cuSparse_CopyToDevice

Call `retval = SUNMatrix_cuSparse_CopyToDevice(A, h_data, h_idxptrs, h_idxvals)`

Description This function copies the matrix information to the GPU device from the provided host arrays. A user may provide `NULL` for any of `h_data`, `h_idxptrs`, or `h_idxvals` to avoid copying that information.

Arguments `A` (`SUNMatrix`)

`h_data` (`realtype*`) a pointer to an allocated array of at least `SUNMatrix_cuSparse_NNZ(A) * sizeof(realtype)` bytes; the nonzero values will be copied from this array onto the device

`h_idxptrs` (`int*`) a pointer to an allocated array of at least `(SUNMatrix_cuSparse_BlockDim(A)+1) * sizeof(int)` bytes; the index pointers will be copied from this array onto the device

`h_idxvals` (`int*`) a pointer to an allocated array of at least `SUNMatrix_cuSparse_BlockNNZ(A) * sizeof(int)` bytes; the index values will be copied from this array onto the device

Return value `SUNMAT_SUCCESS` if the copy operation(s) were successful, or a nonzero error code otherwise.

SUNMatrix_cuSparse_CopyFromDevice

Call `retval = SUNMatrix_cuSparse_CopyFromDevice(A, h_data, h_idxptrs, h_idxvals)`

Description This function copies the matrix information from the GPU device to the provided host arrays. A user may provide `NULL` for any of `h_data`, `h_idxptrs`, or `h_idxvals` to avoid copying that information.

Arguments `A` (`SUNMatrix`)

`h_data` (`realtype*`) a pointer to an allocated array of at least `SUNMatrix_cuSparse_NNZ(A) * sizeof(realtype)` bytes; the nonzero values will be copied into this array from the device

`h_idxptrs` (`int*`) a pointer to an allocated array of at least `(SUNMatrix_cuSparse_BlockDim(A)+1) * sizeof(int)` bytes; the index pointers will be copied into this array from the device

`h_idxvals` (`int*`) a pointer to an allocated array of at least `SUNMatrix_cuSparse_BlockNNZ(A) * sizeof(int)` bytes; the index values will be copied into this array from the device

Return value `SUNMAT_SUCCESS` if the copy operation(s) were successful, or a nonzero error code otherwise.

SUNMatrix_cuSparse_SetKernelExecPolicy

Call `retval = SUNMatrix_cuSparse_SetKernelExecPolicy(A, exec_policy);`

Description This function sets the execution policies which control the kernel parameters utilized when launching the CUDA kernels. By default the matrix is setup to use a policy which tries to leverage the structure of the matrix. See section 7.9.2 for more information about the `SUNCudaExecPolicy` class.

Arguments `A` (`SUNMatrix`)

`exec_policy` (`SUNCudaExecPolicy*`)

Return value `SUNMAT_SUCCESS` if the operation(s) were successful, or a nonzero error code otherwise.

Notes All matrices and vector used in a single instance of a SUNDIALS solver must use the same CUDA stream, and the CUDA stream must be set prior to solver initialization.

SUNMatrix_cuSparse_SetFixedPattern

Call `retval = SUNMatrix_cuSparse_SetFixedPattern(A, yesno)`

Description This function changes the behavior of the `SUNMatZero` operation on the `SUNMatrix` object `A`. By default the matrix sparsity pattern is not considered to be fixed, thus, the `SUNMatZero` operation zeros out all `data` array as well as the `indexvalues` and `indexpointers` arrays. Providing a value of 1 or `SUNTRUE` for the `yesno` argument changes the behavior of `SUNMatZero` on `A` so that only the data is zeroed out, but not the `indexvalues` or `indexpointers` arrays. Providing a value of 0 or `SUNFALSE` for the `yesno` argument is equivalent to the default behavior.

Arguments `A` (`SUNMatrix`)
`yesno` (`booleantype`)

Return value `SUNMAT_SUCCESS` if the operation(s) were successful, or a nonzero error code otherwise.

8.7.2 SUNMatrix_cuSparse Usage Notes

The `SUNMATRIX_CUSPARSE` module only supports 32-bit indexing, thus `SUNDIALS` must be built for 32-bit indexing to use this module.

The `SUNMATRIX_CUSPARSE` module can be used with CUDA streams by calling the `cuSPARSE` function `cusparseSetStream` on the `cusparseHandle_t` that is provided to the `SUNMATRIX_CUSPARSE` constructor.



When using the `SUNMATRIX_CUSPARSE` module with a `SUNDIALS` package (e.g. `CVODE`), the stream given to `cuSPARSE` should be the same stream used for the `NVECTOR` object that is provided to the package, and the `NVECTOR` object given to the `SUNMatvec` operation. If different streams are utilized, synchronization issues may occur.

8.8 The SUNMATRIX_MAGMADENSE implementation

The `SUNMATRIX_MAGMADENSE` implementation of the `SUNDIALS SUNMatrix` API interfaces to the `MAGMA` (`()`) linear algebra library, and can target `NVIDIA`'s `CUDA` programming model or `AMD`'s `HIP` programming model [39]. All data stored by this matrix implementation resides on the GPU at all times. The implementation currently supports a standard `LAPACK` column-major storage format as well as a low-storage format for block-diagonal matrices

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 & 0 & \cdots & 0 \\ 0 & \mathbf{A}_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A}_{n-1} \end{bmatrix}.$$

This matrix implementation is best paired with the `SUNLINEARSOLVER_MAGMADENSE SUNLinearSolver`.

The header file to include when using this module is `sunmatrix/sunmatrix_magmadense.h`. The installed library to link to is `libsundials_sunmatrixmagmadense.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.



The `SUNMATRIX_MAGMADENSE` module is experimental and subject to change.

8.8.1 SUNMATRIX_MAGMADENSE functions

The `SUNMATRIX_MAGMADENSE` module defines GPU-enabled implementations of all matrix operations listed in Section 8.1.1.

1. `SUNMatGetID_MagmaDense` – returns `SUNMATRIX_MAGMADENSE`
2. `SUNMatClone_MagmaDense`
3. `SUNMatDestroy_MagmaDense`
4. `SUNMatZero_MagmaDense`
5. `SUNMatCopy_MagmaDense`
6. `SUNMatScaleAdd_MagmaDense`
7. `SUNMatScaleAddI_MagmaDense`
8. `SUNMatMatvecSetup_MagmaDense`

9. SUNMatMatvec_MagmaDense

10. SUNMatSpace_MagmaDense

In addition, the SUNMATRIX_MAGMADENSE module defines the following implementation specific functions:

SUNMatrix_MagmaDense

Call `A = SUNMatrix_MagmaDense(M, N, memtype, memhelper, queue)`

Description This constructor function creates and allocates memory for an $M \times N$ SUNMATRIX_MAGMADENSE SUNMatrix.

Arguments `M` (`sunindextype`) the number of matrix rows
`N` (`sunindextype`) the number of matrix columns
`memtype` (`SUNMemoryType`) the type of memory to use for the matrix data; can be `SUNMEMTYPE_UVM` or `SUNMEMTYPE_DEVICE`.
`memhelper` (`SUNMemoryHelper`) the memory helper used for allocating data
`queue` a `cudaStream_t` when using CUDA or a `hipStream_t` when using HIP

Return value A SUNMatrix object if successful else NULL.

SUNMatrix_MagmaDenseBlock

Call `A = SUNMatrix_MagmaDenseBlock(nblocks, M_block, N_block, memtype, memhelper, queue)`

Description This constructor function creates and allocates memory for a SUNMATRIX_MAGMADENSE SUNMatrix that is block diagonal with `nblocks` blocks of size $M \times N$.

Arguments `nblocks` (`sunindextype`) the number of matrix blocks
`M_block` (`sunindextype`) the number of matrix rows in each block
`N_block` (`sunindextype`) the number of matrix columns in each block
`memtype` (`SUNMemoryType`) the type of memory to use for the matrix data; can be `SUNMEMTYPE_UVM` or `SUNMEMTYPE_DEVICE`.
`memhelper` (`SUNMemoryHelper`) the memory helper used for allocating data
`queue` a `cudaStream_t` when using CUDA or a `hipStream_t` when using HIP

Return value A SUNMatrix object if successful else NULL.

Notes The block diagonal format currently supports square matrices only.

SUNMatrix_MagmaDense_Rows

Call `M = SUNMatrix_MagmaDense_Rows(A)`

Description This function returns the rows dimension for the $M \times N$ SUNMatrix. For block diagonal matrices, this is computed as $M_{\text{block}} \times \text{nblocks}$.

Arguments `A` (SUNMatrix)

Return value The number of rows in the SUNMatrix.

SUNMatrix_MagmaDense_Columns

Call `N = SUNMatrix_MagmaDense_Columns(A)`

Description This function returns the columns dimension for the $M \times N$ SUNMatrix. For block diagonal matrices, this is computed as $N_{\text{block}} \times \text{nblocks}$.

Arguments `A` (SUNMatrix)

Return value The number of columns in the SUNMatrix.

`SUNMatrix_MagmaDense_BlockRows`

Call `M = SUNMatrix_MagmaDense_BlockRows(A)`

Description This function returns the number of rows in a block of the `SUNMatrix`.

Arguments `A` (`SUNMatrix`)

Return value The number of rows in a block of the `SUNMatrix`.

`SUNMatrix_MagmaDense_BlockColumns`

Call `N = SUNMatrix_MagmaDense_BlockColumns(A)`

Description This function returns the number of columns in a block of the `SUNMatrix`.

Arguments `A` (`SUNMatrix`)

Return value The number of columns in a block of the `SUNMatrix`.

`SUNMatrix_MagmaDense_LData`

Call `ldata = SUNMatrix_MagmaDense_LData(A)`

Description This function returns the length of the data array for the `SUNMatrix`.

Arguments `A` (`SUNMatrix`)

Return value The length of the data array for the `SUNMatrix`.

`SUNMatrix_MagmaDense_NumBlocks`

Call `nblocks = SUNMatrix_MagmaDense_NumBlocks(A)`

Description This function returns the number of blocks in the `SUNMatrix`.

Arguments `A` (`SUNMatrix`)

Return value The number of matrix blocks.

`SUNMatrix_MagmaDense_Data`

Call `data = SUNMatrix_MagmaDense_Data(A)`

Description This function returns the `SUNMatrix` data array.

Arguments `A` (`SUNMatrix`)

Return value An array of pointers to the data arrays for each block in the `SUNMatrix`.

`SUNMatrix_MagmaDense_BlockData`

Call `data = SUNMatrix_MagmaDense_BlockData(A)`

Description This function returns an array of pointers that point to the start of the data array for each block.

Arguments `A` (`SUNMatrix`)

Return value An array of pointers to the data arrays for each block in the `SUNMatrix`.

`SUNMatrix_MagmaDense_Block`

Call `data = SUNMatrix_MagmaDense_Block(A, k)`

Description This function returns a pointer to the data for block k .

Arguments `A` (`SUNMatrix`)

Return value A pointer to the start of the data array for block k in the `SUNMatrix`.

Notes No bounds-checking is performed, k should be strictly less than `nblocks`.

SUNMatrix_MagmaDense_Column

Call `data = SUNMatrix_MagmaDense_Column(A, j)`

Description This function returns a pointer to the data for column j of the matrix.

Arguments `A` (`SUNMatrix`)

Return value A pointer to the start of the data array for column j of the `SUNMatrix`.

Notes No bounds-checking is performed, j should be strictly less than $n_{\text{blocks}} * N_{\text{block}}$.

SUNMatrix_MagmaDense_BlockColumn

Call `data = SUNMatrix_MagmaDense_Column(A, k, j)`

Description This function returns a pointer to the data for column j of block k .

Arguments `A` (`SUNMatrix`)

Return value A pointer to the start of the data array for column j of block k in the `SUNMatrix`.

Notes No bounds-checking is performed.

SUNMatrix_MagmaDense_CopyToDevice

Call `retval = SUNMatrix_MagmaDense_CopyToDevice(A, h_data)`

Description This functions copies the matrix data to the GPU device from the provided host array.

Arguments `A` (`SUNMatrix`)

`h_data` (`realtype*`)

Return value `SUNMAT_SUCCESS` if the copy operation was successful, or a nonzero error code otherwise

SUNMatrix_MagmaDense_CopyFromDevice

Call `retval = SUNMatrix_MagmaDense_CopyFromDevice(A, h_data)`

Description This functions copies the matrix data from the GPU device to the provided host array.

Arguments `A` (`SUNMatrix`)

`h_data` (`realtype*`)

Return value `SUNMAT_SUCCESS` if the copy operation was successful, or a nonzero error code otherwise

8.8.2 SUNMATRIX_MAGMADENSE Usage Notes

When using the `SUNMATRIX_MAGMADENSE` module with a `SUNDIALS` package (e.g. `CVODE`), the stream given to matrix should be the same stream used for the `NVECTOR` object that is provided to the package, and the `NVECTOR` object given to the `SUNMatvec` operation. If different streams are utilized, synchronization issues may occur.

**8.9 The SUNMATRIX_ONEMKLDENSE implementation**

The `SUNMATRIX_ONEMKLDENSE` implementation of the `SUNMatrix` class is intended for interfacing with direct linear solvers from the [Intel oneAPI Math Kernel Library \(oneMKL\)](#) using the SYCL (DPC++) programming model. The implementation currently supports a standard LAPACK column-major storage format as well as a low-storage format for block-diagonal matrices

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 & 0 & \cdots & 0 \\ 0 & \mathbf{A}_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A}_{n-1} \end{bmatrix}$$

This matrix implementation is best paired with the `SUNLINEARSOLVER_ONEMKLDENSE` `SUNLinearSolver`.

The header file to include when using this class is `sunmatrix/sunmatrix_onemkldense.h`. The installed library to link to is `libsundials_sunmatrixonekkldense.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.



The `SUNMATRIX_ONEMKLDENSE` class is experimental and subject to change.

8.9.1 SUNMATRIX_ONEMKLDENSE functions

The `SUNMATRIX_ONEMKLDENSE` class defines implementations of the following matrix operations listed in Section 8.1.1.

1. `SUNMatGetID_OneMkldense` – returns `SUNMATRIX_ONEMKLDENSE`
2. `SUNMatClone_OneMkldense`
3. `SUNMatDestroy_OneMkldense`
4. `SUNMatZero_OneMkldense`
5. `SUNMatCopy_OneMkldense`
6. `SUNMatScaleAdd_OneMkldense`
7. `SUNMatScaleAddI_OneMkldense`
8. `SUNMatMatvec_OneMkldense`
9. `SUNMatSpace_OneMkldense`

In addition, the `SUNMATRIX_ONEMKLDENSE` class class defines the following implementation specific functions.

Constructors

<code>SUNMatrix_OneMkldense</code>

Call `A = SUNMatrix_OneMkldense(M, N, memtype, memhelper, queue)`

Description This constructor function creates and allocates memory for an $M \times N$ `SUNMATRIX_ONEMKLDENSE` `SUNMatrix`.

Arguments

<code>M</code>	(<code>sunindextype</code>) the number of matrix rows
<code>N</code>	(<code>sunindextype</code>) the number of matrix columns
<code>memtype</code>	(<code>SUNMemoryType</code>) the type of memory to use for the matrix data; can be <code>SUNMEMTYPE_UVM</code> or <code>SUNMEMTYPE_DEVICE</code> .
<code>memhelper</code>	(<code>SUNMemoryHelper</code>) the memory helper used for allocating data
<code>queue</code>	(<code>sycl::queue*</code>) the SYCL queue to which operation will be submitted

Return value A `SUNMatrix` object if successful else `NULL`.

SUNMatrix_OneMklDenseBlock

Call `A = SUNMatrix_OneMklDenseBlock(nblocks, M_block, N_block, memtype, memhelper, queue)`

Description This constructor function creates and allocates memory for a `SUNMATRIX_ONEMKLDENSE SUNMatrix` that is block diagonal with `nblocks` blocks of size $M_{block} \times N_{block}$.

Arguments `nblocks` (`sunindextype`) the number of matrix blocks
`M_block` (`sunindextype`) the number of matrix rows in each block
`N_block` (`sunindextype`) the number of matrix columns in each block
`memtype` (`SUNMemoryType`) the type of memory to use for the matrix data; can be `SUNMEMTYPE_UVM` or `SUNMEMTYPE_DEVICE`.
`memhelper` (`SUNMemoryHelper`) the memory helper used for allocating data
`queue` (`sycl::queue*`) the SYCL queue to which operation will be submitted

Return value A `SUNMatrix` object if successful else `NULL`.

Notes The block diagonal format currently supports square matrices only.

Access Matrix Dimensions**SUNMatrix_OneMklDense_Rows**

Call `M = SUNMatrix_OneMklDense_Rows(A)`

Description This function returns the rows dimension for the $M \times N$ `SUNMatrix`. For block diagonal matrices, this is computed as $M_{block} \times nblocks$.

Arguments `A` (`SUNMatrix`)

Return value The number of rows in the `SUNMatrix`.

SUNMatrix_OneMklDense_Columns

Call `N = SUNMatrix_OneMklDense_Columns(A)`

Description This function returns the columns dimension for the $M \times N$ `SUNMatrix`. For block diagonal matrices, this is computed as $N_{block} \times nblocks$.

Arguments `A` (`SUNMatrix`)

Return value The number of columns in the `SUNMatrix`.

Access Matrix Block Dimensions**SUNMatrix_OneMklDense_NumBlocks**

Call `nblocks = SUNMatrix_OneMklDense_NumBlocks(A)`

Description This function returns the number of blocks in the `SUNMatrix`.

Arguments `A` (`SUNMatrix`)

Return value The number of matrix blocks.

SUNMatrix_OneMklDense_BlockRows

Call `M = SUNMatrix_OneMklDense_BlockRows(A)`

Description This function returns the number of rows in a block of the `SUNMatrix`.

Arguments `A` (`SUNMatrix`)

Return value The number of rows in a block of the `SUNMatrix`.

`SUNMatrix_OneMklDense_BlockColumns`

Call `N = SUNMatrix_OneMklDense_BlockColumns(A)`

Description This function returns the number of columns in a block of the `SUNMatrix`.

Arguments `A` (`SUNMatrix`)

Return value The number of columns in a block of the `SUNMatrix`.

Access Matrix Data`SUNMatrix_OneMklDense_LData`

Call `ldata = SUNMatrix_OneMklDense_LData(A)`

Description This function returns the length of the data array for the `SUNMatrix`.

Arguments `A` (`SUNMatrix`)

Return value The length of the data array for the `SUNMatrix`.

`SUNMatrix_OneMklDense_Data`

Call `data = SUNMatrix_OneMklDense_Data(A)`

Description This function returns the `SUNMatrix` data array.

Arguments `A` (`SUNMatrix`)

Return value An array of pointers to the data arrays for each block in the `SUNMatrix`.

`SUNMatrix_OneMklDense_Column`

Call `data = SUNMatrix_OneMklDense_Column(A, j)`

Description This function returns a pointer to the data for column j of the matrix.

Arguments `A` (`SUNMatrix`)

Return value A pointer to the start of the data array for column j of the `SUNMatrix`.

Notes No bounds-checking is performed, j should be strictly less than $n_{\text{blocks}} * N_{\text{block}}$.

Access Matrix Data`SUNMatrix_OneMklDense_BlockLData`

Call `ldata = SUNMatrix_OneMklDense_BlockLData(A)`

Description This function returns the length of the data array for the `SUNMatrix` for each block of the `SUNMatrix` object.

Arguments `A` (`SUNMatrix`)

Return value The length of the data array for each block of the `SUNMatrix`.

`SUNMatrix_OneMklDense_BlockData`

Call `data = SUNMatrix_OneMklDense_BlockData(A)`

Description This function returns an array of pointers that point to the start of the data array for each block.

Arguments `A` (`SUNMatrix`)

Return value An array of pointers to the data arrays for each block in the `SUNMatrix`.

SUNMatrix_OneMklDense_Block

Call `data = SUNMatrix_OneMklDense_Block(A, k)`

Description This function returns a pointer to the data for block k .

Arguments `A` (`SUNMatrix`)

Return value A pointer to the start of the data array for block k in the `SUNMatrix`.

Notes No bounds-checking is performed, k should be strictly less than `nblocks`.

SUNMatrix_OneMklDense_BlockColumn

Call `data = SUNMatrix_OneMklDense_Column(A, k, j)`

Description This function returns a pointer to the data for column j of block k .

Arguments `A` (`SUNMatrix`)

Return value A pointer to the start of the data array for column j of block k in the `SUNMatrix`.

Notes No bounds-checking is performed.

Copy Data**SUNMatrix_OneMklDense_CopyToDevice**

Call `retval = SUNMatrix_OneMklDense_CopyToDevice(A, h_data)`

Description This function copies the matrix data to the GPU device from the provided host array.

Arguments `A` (`SUNMatrix`)
`h_data` (`realtype*`)

Return value `SUNMAT_SUCCESS` if the copy operation was successful, or a nonzero error code otherwise

SUNMatrix_OneMklDense_CopyFromDevice

Call `retval = SUNMatrix_OneMklDense_CopyFromDevice(A, h_data)`

Description This function copies the matrix data from the GPU device to the provided host array.

Arguments `A` (`SUNMatrix`)
`h_data` (`realtype*`)

Return value `SUNMAT_SUCCESS` if the copy operation was successful, or a nonzero error code otherwise

8.9.2 SUNMATRIX_ONEMKLDENSE Usage Notes

The `SUNMATRIX_ONEMKLDENSE` class only supports 64-bit indexing, thus `SUNDIALS` must be built for 64-bit indexing to use this class.

When using the `SUNMATRIX_ONEMKLDENSE` class with a `SUNDIALS` package (e.g. `CVODE`), the stream given to matrix should be the same stream used for the `NVECTOR` object that is provided to the package, and the `NVECTOR` object given to the `SUNMatvec` operation. If different streams are utilized, synchronization issues may occur.



Chapter 9

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 iterative (matrix-based or matrix-free) 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, user-supplied implementations for matrix-based iterative linear solvers and linear solvers with ‘embedded’ matrices are also 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

$$\tilde{A}\tilde{x} = \tilde{b} \tag{9.1}$$

where

$$\begin{aligned} \tilde{A} &= S_1 P_1^{-1} A P_2^{-1} S_2^{-1}, \\ \tilde{b} &= S_1 P_1^{-1} b, \\ \tilde{x} &= S_2 P_2 x, \end{aligned} \tag{9.2}$$

and where

- P_1 is the left preconditioner,
- P_2 is the right preconditioner,
- S_1 is a diagonal matrix of scale factors for $P_1^{-1}b$,
- S_2 is a diagonal matrix of scale factors for P_2x .

The scaling matrices are chosen so that $S_1 P_1^{-1} b$ and $S_2 P_2 x$ have dimensionless components. If preconditioning is done on the left only ($P_2 = I$), by a matrix P , then S_2 must be a scaling for x , while S_1 is a scaling for $P^{-1} b$, and so may also be taken as a scaling for x . Similarly, if preconditioning is done on the right only ($P_1 = I$ and $P_2 = P$), then S_1 must be a scaling for b , while S_2 is a scaling for Px , and may also be taken as a scaling for b .

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

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

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 (see §9.4.2 for more details). In this case, they instead request that iterative linear solvers stop based on the criteria

$$\left\| P_1^{-1} b - P_1^{-1} A x \right\|_2 < \text{tol.}$$

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.

For users interested in providing their own SUNLINSOL module, the following section presents the SUNLINSOL API and its implementation beginning with the definition of SUNLINSOL functions in sections 9.1.1 – 9.1.3. This is followed by the definition of functions supplied to a linear solver implementation in section 9.1.4. A table of linear solver return codes is given in section 9.1.5. The SUNLinearSolver type and the generic SUNLINSOL module are defined in section 9.1.6. The section 9.2 discusses compatibility between the SUNDIALS-provided SUNLINSOL modules and SUNMATRIX modules. Section 9.3 lists the requirements for supplying a custom SUNLINSOL module and discusses some intended use cases. Users wishing to supply their own SUNLINSOL module are encouraged to use the SUNLINSOL implementations provided with SUNDIALS as a template for supplying custom linear solver modules. The SUNLINSOL functions required by this SUNDIALS package as well as other package specific details are given in section 9.4. The remaining sections of this chapter present the SUNLINSOL modules provided with SUNDIALS.

9.1 The SUNLinearSolver API

The SUNLINSOL API defines several linear solver operations that enable SUNDIALS packages to utilize any SUNLINSOL implementation that provides the required functions. These functions can be divided into three categories. The first are the core linear solver functions. The second group of functions consists of set routines to supply the linear solver with functions provided by the SUNDIALS time integrators and to modify solver parameters. The final group consists of get routines for retrieving linear solver statistics. All of these functions are defined in the header file `sundials/sundials_linearsolver.h`.

9.1.1 SUNLinearSolver core functions

The core linear solver functions consist of two required functions to get the linear solver type (`SUNLinSolGetType`) and solve the linear system $Ax = b$ (`SUNLinSolSolve`). The remaining functions are for getting the solver ID (`SUNLinSolGetID`), initializing the linear solver object once all solver-specific options have been set (`SUNLinSolInitialize`), setting up the linear solver object to utilize an updated matrix A (`SUNLinSolSetup`), and for destroying the linear solver object (`SUNLinSolFree`) are optional.

SUNLinSolGetType

Call `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, iterative, or matrix-iterative) from the abstract `SUNLinearSolver` interface.

Arguments LS (`SUNLinearSolver`) a `SUNLINSOL` object.

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

- `SUNLINEARSOLVER_DIRECT` – 0, the `SUNLINSOL` module requires a matrix, and computes an ‘exact’ solution to the linear system defined by that matrix.
- `SUNLINEARSOLVER_ITERATIVE` – 1, the `SUNLINSOL` module does not require a matrix (though one may be provided), and computes an inexact solution to the linear system using a matrix-free iterative algorithm. That is it solves the linear system defined by the package-supplied `ATimes` routine (see `SUNLinSolSetATimes` below), even if that linear system differs from the one encoded in the matrix object (if one is provided). As the solver computes the solution only inexactly (or may diverge), the linear solver should check for solution convergence/accuracy as appropriate.
- `SUNLINEARSOLVER_MATRIX_ITERATIVE` – 2, the `SUNLINSOL` module requires a matrix, and computes an inexact solution to the linear system defined by that matrix using an iterative algorithm. That is it solves the linear system defined by the matrix object even if that linear system differs from that encoded by the package-supplied `ATimes` routine. As the solver computes the solution only inexactly (or may diverge), the linear solver should check for solution convergence/accuracy as appropriate.
- `SUNLINEARSOLVER_MATRIX_EMBEDDED` – 3, the `SUNLINSOL` module sets up and solves the specified linear system at each linear solve call. Any matrix-related data structures are held internally to the linear solver itself, and are not provided by the `SUNDIALS` package.

Notes See section 9.3.1 for more information on intended use cases corresponding to the linear solver type.

F2003 Name `FSUNLinSolGetType`

SUNLinSolGetID

Call `id = SUNLinSolGetID(LS);`

Description The *optional* function `SUNLinSolGetID` returns the identifier for the linear solver LS.

Arguments LS (`SUNLinearSolver`) a `SUNLINSOL` object.

Return value The return value `id` (of type `int`) will be a non-negative value defined by the enumeration `SUNLinearSolver_ID`. The possible enumeration values are specified in the `sundials_linearsolver.h` header file.

Notes It is recommended that a user-supplied `SUNLinearSolver` return the `SUNLINEARSOLVER_CUSTOM` identifier.

F2003 Name `FSUNLinSolGetID`

SUNLinSolInitialize

Call `retval = SUNLinSolInitialize(LS);`

Description The *optional* function `SUNLinSolInitialize` performs linear solver initialization (assuming 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 9.1.

F2003 Name FSUNLinSolInitialize

SUNLinSolSetup

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

Description The *optional* 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 9.1.

F2003 Name FSUNLinSolSetup

SUNLinSolSolve

Call `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 containing the initial guess for the solution of the linear system, and the solution to the linear system upon return.

b (`N_Vector`) a NVECTOR object containing the linear system right-hand side.

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 9.1.

Notes **Direct solvers:** can ignore the `tol` argument.

Matrix-free solvers: (those that identify as `SUNLINEARSOLVER_ITERATIVE`) can ignore the SUNMATRIX input `A`, and should instead rely on the matrix-vector product function supplied through the routine `SUNLinSolSetATimes`.

Iterative solvers: (those that identify as `SUNLINEARSOLVER_ITERATIVE` or `SUNLINEARSOLVER_MATRIX_ITERATIVE`) 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.

Matrix-embedded solvers: should ignore the SUNMATRIX input `A` as this will be NULL. It is assumed that within this call, the solver will call interface routines from the relevant SUNDIALS package to directly form the relevant linear system matrix `A`, and then solve the system before returning with the solution `x`.

F2003 Name FSUNLinSolSolve

SUNLinSolFree

Call `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.

F2003 Name `FSUNLinSolFree`

9.1.2 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 even then is only required 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.

SUNLinSolSetATimes

Call `retval = SUNLinSolSetATimes(LS, A_data, ATimes);`

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 9.1.

F2003 Name `FSUNLinSolSetATimes`

SUNLinSolSetPreconditioner

Call `retval = SUNLinSolSetPreconditioner(LS, Pdata, Pset, 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 (9.1)-(9.2). 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.

`Pdata` (`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 9.1.

F2003 Name `FSUNLinSolSetPreconditioner`

SUNLinSolSetScalingVectors

Call `retval = SUNLinSolSetScalingVectors(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 (9.1)-(9.2), 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 9.1.

F2003 Name `FSUNLinSolSetScalingVectors`

SUNLinSolSetZeroGuess

Call `retval = SUNLinSolSetZeroGuess(LS, onoff);`

Description The *optional* function `SUNLinSolSetZeroGuess` indicates if the next call to `SUNLinSolSolve` will be made with a zero initial guess.

Arguments `LS` (`SUNLinearSolver`) a `SUNLINSOL` object.

`onoff` (`booleantype`) a flag indicating if the initial guess to linear solver is zero (`SUNTRUE`) or non-zero (`SUNFALSE`).

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 9.1.

Notes It is assumed that the initial guess status is not retained across calls to `SUNLinSolSolve`. As such, the linear solver interfaces in each of the `SUNDIALS` packages call `SUNLinSolSetZeroGuess` prior to each call to `SUNLinSolSolve`.

F2003 Name `FSUNLinSolSetZeroGuess`

9.1.3 SUNLinearSolver get functions

The following get functions allow `SUNDIALS` packages to retrieve results from a linear solve. All routines are optional.

SUNLinSolNumIters

Call `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

F2003 Name `FSUNLinSolNumIters`

SUNLinSolResNorm

Call `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

F2003 Name `FSUNLinSolResNorm`

SUNLinSolResid

Call `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 set to `NULL` in the implementation.

F2003 Name `FSUNLinSolResid`

SUNLinSolLastFlag

Call `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 `sunindextype` containing the most recent error flag

F2003 Name `FSUNLinSolLastFlag`

SUNLinSolSpace

Call `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.

`liw` (`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 9.1.

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

F2003 Name `FSUNLinSolSpace`

9.1.4 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.

ATimesFn

Definition `typedef int (*ATimesFn)(void *A.data, N_Vector v, N_Vector z);`

Purpose These functions compute the action of a matrix on a vector, performing the operation $z = Av$. Memory for z should already be allocated prior to calling this function. The vector v should be left unchanged.

Arguments `A_data` is a pointer to client data, the same as that supplied to `SUNLinSolSetATimes`.
`v` is the input vector to multiply.
`z` is the output vector computed.

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

PSetupFn

Definition `typedef int (*PSetupFn)(void *P.data)`

Purpose These functions set up any requisite problem data in preparation for calls to the corresponding `PSolveFn`.

Arguments `P_data` is a pointer to client data, the same pointer as that supplied to the routine `SUNLinSolSetPreconditioner`.

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

PSolveFn

Definition `typedef int (*PSolveFn)(void *P.data, N_Vector r, N_Vector z, realtype tol, int lr)`

Purpose These functions solve the preconditioner equation $Pz = r$ for the vector z . Memory for z should already be allocated prior to calling this function. The parameter `P_data` is a pointer to any information about P which the function needs in order to do its job (set up by the corresponding `PSetupFn`). The parameter `lr` is input, and indicates whether P is to be taken as the left preconditioner or the right preconditioner: `lr = 1` for left and `lr = 2` for right. If preconditioning is on one side only, `lr` can be ignored. If the preconditioner is iterative, then it should strive to solve the preconditioner equation so that

$$\|Pz - r\|_{\text{wrms}} < \text{tol}$$

where the weight vector for the WRMS norm may be accessed from the main package memory structure. The vector r should not be modified by the `PSolveFn`.

Arguments `P_data` is a pointer to client data, the same pointer as that supplied to the routine `SUNLinSolSetPreconditioner`.

`r` is the right-hand side vector for the preconditioner system.

`z` is the solution vector for the preconditioner system.

`tol` is the desired tolerance for an iterative preconditioner.

`lr` is flag indicating whether the routine should perform left (1) or right (2) preconditioning.

Return value This routine should return 0 if successful and a non-zero value if unsuccessful. On a failure, a negative return value indicates an unrecoverable condition, while a positive value indicates a recoverable one, in which the calling routine may reattempt the solution after updating preconditioner data.

9.1.5 SUNLinearSolver return codes

The functions provided to SUNLINSOL modules by each SUNDIALS package, and functions within the SUNDIALS-provided SUNLINSOL implementations utilize a common set of return codes, shown in Table 9.1. 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 9.1: Description of the SUNLinearSolver error codes

Name	Value	Description
SUNLS_SUCCESS	0	successful call or converged solve
SUNLS_MEM_NULL	-801	the memory argument to the function is NULL
SUNLS_ILL_INPUT	-802	an illegal input has been provided to the function
SUNLS_MEM_FAIL	-803	failed memory access or allocation
SUNLS_ATIMES_NULL	-804	the Atimes function is NULL
SUNLS_ATIMES_FAIL_UNREC	-805	an unrecoverable failure occurred in the ATimes routine
SUNLS_PSET_FAIL_UNREC	-806	an unrecoverable failure occurred in the Pset routine
SUNLS_PSOLVE_NULL	-807	the preconditioner solve function is NULL
SUNLS_PSOLVE_FAIL_UNREC	-808	an unrecoverable failure occurred in the Psolve routine
SUNLS_PACKAGE_FAIL_UNREC	-809	an unrecoverable failure occurred in an external linear solver package
SUNLS_GS_FAIL	-810	a failure occurred during Gram-Schmidt orthogonalization (SUNLINSOL_SPGMR/SUNLINSOL_SPGFMR)
SUNLS_QRSOL_FAIL	-811	a singular R matrix was encountered in a QR factorization (SUNLINSOL_SPGMR/SUNLINSOL_SPGFMR)
SUNLS_VECTOROP_ERR	-812	a vector operation error occurred
SUNLS_RES_REDUCED	801	an iterative solver reduced the residual, but did not converge to the desired tolerance
SUNLS_CONV_FAIL	802	an iterative solver did not converge (and the residual was not reduced)
SUNLS_ATIMES_FAIL_REC	803	a recoverable failure occurred in the ATimes routine
SUNLS_PSET_FAIL_REC	804	a recoverable failure occurred in the Pset routine
SUNLS_PSOLVE_FAIL_REC	805	a recoverable failure occurred in the Psolve routine
SUNLS_PACKAGE_FAIL_REC	806	a recoverable failure occurred in an external linear solver package

continued on next page

Name	Value	Description
SUNLS_QRFACT_FAIL	807	a singular matrix was encountered during a QR factorization (SUNLINSOL_SPGMR/SUNLINSOL_SPPGMR)
SUNLS_LUFACT_FAIL	808	a singular matrix was encountered during a LU factorization (SUNLINSOL_DENSE/SUNLINSOL_BAND)

9.1.6 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);
    SUNLinearSolver_ID (*getid)(SUNLinearSolver);
    int (*setatimes)(SUNLinearSolver, void*, ATimesFn);
    int (*setpreconditioner)(SUNLinearSolver, void*,
                            PSetupFn, PSolveFn);
    int (*setscalingvectors)(SUNLinearSolver,
                             N_Vector, N_Vector);
    int (*setzeroguess)(SUNLinearSolver, booleantype);
    int (*initialize)(SUNLinearSolver);
    int (*setup)(SUNLinearSolver, SUNMatrix);
    int (*solve)(SUNLinearSolver, SUNMatrix, N_Vector,
                 N_Vector, realtype);
    int (*numiters)(SUNLinearSolver);
    realtype (*resnorm)(SUNLinearSolver);
    sunindxetype (*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 9.1.1-9.1.3. 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:

```

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

```

The Fortran 2003 interface provides a `bind(C)` derived-type for the `_generic_SUNLinearSolver` and the `_generic_SUNLinearSolver_Ops` structures. Their definition is given below.

```

type, bind(C), public :: SUNLinearSolver
  type(C_PTR), public :: content
  type(C_PTR), public :: ops
end type SUNLinearSolver

type, bind(C), public :: SUNLinearSolver_Ops
  type(C_FUNPTR), public :: gettype
  type(C_FUNPTR), public :: setatimes
  type(C_FUNPTR), public :: setpreconditioner
  type(C_FUNPTR), public :: setscalingvectors
  type(C_FUNPTR), public :: setzeroguess
  type(C_FUNPTR), public :: initialize
  type(C_FUNPTR), public :: setup
  type(C_FUNPTR), public :: solve
  type(C_FUNPTR), public :: numiters
  type(C_FUNPTR), public :: resnorm
  type(C_FUNPTR), public :: lastflag
  type(C_FUNPTR), public :: space
  type(C_FUNPTR), public :: resid
  type(C_FUNPTR), public :: free
end type SUNLinearSolver_Ops

```

9.2 Compatibility of SUNLinearSolver modules

We note that not all SUNLINSOL types are compatible with all SUNMATRIX and NVECTOR types provided with SUNDIALS. In Table 9.2 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 9.2: SUNDIALS matrix-based linear solvers and matrix implementations that can be used for each.

Linear Solver Interface	Dense Matrix	Banded Matrix	Sparse Matrix	SLUNRloc Matrix	User Supplied
Dense	✓				✓
Band		✓			✓
LapackDense	✓				✓
LapackBand		✓			✓
KLU			✓		✓
SuperLU_DIST				✓	✓
SUPERLUMT			✓		✓
<i>continued on next page</i>					

Linear Solver Interface	Dense Matrix	Banded Matrix	Sparse Matrix	SLUNRloc Matrix	User Supplied
User supplied	✓	✓	✓	✓	✓

9.3 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 section 9.4 to determine which SUNLINSOL operations are required for this SUNDIALS package.

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.

To aid in the creation of custom SUNLINSOL modules the generic SUNLINSOL module provides the utility functions `SUNLinSolNewEmpty` and `SUNLinSolFreeEmpty`. When used in custom SUNLINSOL constructors the function `SUNLinSolNewEmpty` will ease the introduction of any new optional linear solver operations to the SUNLINSOL API by ensuring only required operations need to be set.

`SUNLinSolNewEmpty`

Call `LS = SUNLinSolNewEmpty();`

Description The function `SUNLinSolNewEmpty` allocates a new generic SUNLINSOL object and initializes its content pointer and the function pointers in the operations structure to NULL.

Arguments None

Return value This function returns a `SUNLinearSolver` object. If an error occurs when allocating the object, then this routine will return NULL.

F2003 Name `FSUNLinSolNewEmpty`

`SUNLinSolFreeEmpty`

Call `SUNLinSolFreeEmpty(LS);`

Description This routine frees the generic `SUNLinSolFreeEmpty` object, under the assumption that any implementation-specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the ops pointer is NULL, and, if it is not, it will free it as well.

Arguments `LS (SUNLinearSolver)`

Return value None

F2003 Name `FSUNLinSolFreeEmpty`

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.

9.3.1 Intended use cases

The SUNLINSOL (and SUNMATRIX) APIs are designed to require a minimal set of routines to ease interfacing with custom or third-party linear solver libraries. External solvers provide similar routines with the necessary functionality and thus will require minimal effort to wrap within custom SUNMATRIX and SUNLINSOL implementations. Sections 8.2 and 9.4 include a list of the required set of routines that compatible SUNMATRIX and SUNLINSOL implementations must provide. As SUNDIALS packages utilize generic SUNLINSOL modules allowing for user-supplied `SUNLinearSolver` implementations, there exists a wide range of possible linear solver combinations. Some intended use cases for both the SUNDIALS-provided and user-supplied SUNLINSOL modules are discussed in the following sections.

Direct linear solvers

Direct linear solver modules require a matrix and compute an ‘exact’ solution to the linear system *defined by the matrix*. Multiple matrix formats and associated direct linear solvers are supplied with SUNDIALS through different SUNMATRIX and SUNLINSOL implementations. SUNDIALS packages strive to amortize the high cost of matrix construction by reusing matrix information for multiple nonlinear iterations. As a result, each package’s linear solver interface recomputes Jacobian information as infrequently as possible.

Alternative matrix storage formats and compatible linear solvers that are not currently provided by, or interfaced with, SUNDIALS can leverage this infrastructure with minimal effort. To do so, a user must implement custom SUNMATRIX and SUNLINSOL wrappers for the desired matrix format and/or linear solver following the APIs described in Chapters 8 and 9. *This user-supplied SUNLINSOL module must then self-identify as having `SUNLINEARSOLVER_DIRECT` type.*

Matrix-free iterative linear solvers

Matrix-free iterative linear solver modules do not require a matrix and compute an inexact solution to the linear system *defined by the package-supplied `ATimes` routine*. SUNDIALS supplies multiple scaled, preconditioned iterative linear solver (spils) SUNLINSOL modules that support scaling to allow users to handle non-dimensionalization (as best as possible) within each SUNDIALS package and retain variables and define equations as desired in their applications. For linear solvers that do not support left/right scaling, the tolerance supplied to the linear solver is adjusted to compensate (see section 9.4.2 for more details); however, this use case may be non-optimal and cannot handle situations where the magnitudes of different solution components or equations vary dramatically within a single problem.

To utilize alternative linear solvers that are not currently provided by, or interfaced with, SUNDIALS a user must implement a custom SUNLINSOL wrapper for the linear solver following the API described in Chapter 9. *This user-supplied SUNLINSOL module must then self-identify as having `SUNLINEARSOLVER_ITERATIVE` type.*

Matrix-based iterative linear solvers (reusing A)

Matrix-based iterative linear solver modules require a matrix and compute an inexact solution to the linear system *defined by the matrix*. This matrix will be updated infrequently and reused across multiple solves to amortize cost of matrix construction. As in the direct linear solver case, only wrappers for the matrix and linear solver in SUNMATRIX and SUNLINSOL implementations need to be created to utilize a new linear solver. *This user-supplied SUNLINSOL module must then self-identify as having `SUNLINEARSOLVER_MATRIX_ITERATIVE` type.*

At present, SUNDIALS has one example problem that uses this approach for wrapping a structured-grid matrix, linear solver, and preconditioner from the *hypr* library that may be used as a template for other customized implementations (see `examples/arkode/CXX_parhyp/ark_heat2D_hypr.cpp`).

Matrix-based iterative linear solvers (current A)

For users who wish to utilize a matrix-based iterative linear solver module where the matrix is *purely for preconditioning* and the linear system is *defined by the package-supplied `ATimes` routine*, we envision two current possibilities.

The preferred approach is for users to employ one of the SUNDIALS spils SUNLINSOL implementations (SUNLINSOL_SPGMR, SUNLINSOL_SPCGMR, SUNLINSOL_SPCG, SUNLINSOL_SPTFQMR, or SUNLINSOL_PCG) as the outer solver. The creation and storage of the preconditioner matrix, and interfacing with the corresponding linear solver, can be handled through a package's preconditioner 'setup' and 'solve' functionality (see §4.5.4.2) without creating SUNMATRIX and SUNLINSOL implementations. This usage mode is recommended primarily because the SUNDIALS-provided spils modules support the scaling as described above.

A second approach supported by the linear solver APIs is as follows. If the SUNLINSOL implementation is matrix-based, *self-identifies as having `SUNLINEARSOLVER_ITERATIVE` type*, and *also provides a non-NULL `SUNLinSolSetATimes` routine*, then each SUNDIALS package will call that routine to attach its package-specific matrix-vector product routine to the SUNLINSOL object. The SUNDIALS package will then call the SUNLINSOL-provided `SUNLinSolSetup` routine (infrequently) to update matrix information, but will provide current matrix-vector products to the SUNLINSOL implementation through the package-supplied `ATimesFn` routine.

Application-specific linear solvers with embedded matrix structure

Many applications can exploit additional linear system structure due to the implicit couplings in their model equations. In certain circumstances, the linear solve $Ax = b$ may be performed without the need for a global system matrix A , as the unformed A may be block diagonal or block triangular, and thus the overall linear solve may be performed through a sequence of smaller linear solves. In other circumstances, a linear system solve may be accomplished via specialized fast solvers, such as the fast Fourier transform, fast multipole method, or treecode, in which case no matrix structure may be explicitly necessary. Furthermore, in many of these situations construction and preprocessing of the linear system matrix A may be inexpensive, and thus increased performance may be possible if the current linear system information is used within every solve (instead of being lagged, as occurs with matrix-based solvers that reuse A).

To support such application-specific situations, SUNDIALS supports user-provided linear solvers with the `SUNLINEARSOLVER_MATRIX_EMBEDDED` type. For an application to leverage this support, it should define a custom SUNLINSOL implementation having this type. For this implementation, only the required `SUNLinSolGetType` and `SUNLinSolSolve` operations should be needed. Within `SUNLinSolSolve`, the linear solver implementation should call package-specific interface routines (e.g., `ARKStepGetNonlinearSystemData`, `CVodeGetNonlinearSystemData`, `IDAGetNonlinearSystemData`, `ARKStepGetCurrentGamma`, `CVodeGetCurrentGamma`, `IDAGetCurrentCj` or `MRISetGetCurrentGamma`) to construct the relevant system matrix A (or portions thereof), solve the linear system $Ax = b$, and return the solution vector x .

We note that when attaching this custom SUNLINSOL object with the relevant SUNDIALS package `SetLinearSolver` routine, the input SUNMATRIX A should be set to `NULL`.

9.4 KINSOL SUNLinearSolver interface

Table 9.3 below lists the SUNLINSOL module linear solver functions used within the KINLS interface. 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. Note:

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.
4. Although KINLS does not call `SUNLinSolFree` directly, this routine should be available for users to call when cleaning up from a simulation.

Table 9.3: List of linear solver function usage in the KINLS interface

	DIRECT	ITERATIVE	MATRIX_ITERATIVE
<code>SUNLinSolGetType</code>	✓	✓	✓
<code>SUNLinSolSetATimes</code>	†	✓	†
<code>SUNLinSolSetPreconditioner</code>	†	†	†
<code>SUNLinSolSetScalingVectors</code>	†	†	†
<code>SUNLinSolInitialize</code>	✓	✓	✓
<code>SUNLinSolSetup</code>	✓	✓	✓
<code>SUNLinSolSolve</code>	✓	✓	✓
¹ <code>SUNLinSolNumIters</code>		†	†
² <code>SUNLinSolResNorm</code>		†	†
³ <code>SUNLinSolLastFlag</code>			
⁴ <code>SUNLinSolFree</code>			
<code>SUNLinSolSpace</code>	†	†	†

Since there are a wide range of potential SUNLINSOL use cases, the following subsections describe some details of the KINLS interface, in the case that interested users wish to develop custom SUNLINSOL modules.

9.4.1 Lagged matrix information

If the SUNLINSOL object self-identifies as having type `SUNLINEARSOLVER_DIRECT` or `SUNLINEARSOLVER_MATRIX_ITERATIVE`, then the SUNLINSOL object solves a linear system *defined* by a `SUNMATRIX` object. As a result, KINSOL can perform its optional residual monitoring scheme, described in §2.

9.4.2 Iterative linear solver tolerance

If the SUNLINSOL object self-identifies as having type `SUNLINEARSOLVER_ITERATIVE` or `SUNLINEARSOLVER_MATRIX_ITERATIVE` then KINLS will adjust the linear solver tolerance `delta` as described in §2 during the course of the nonlinear solve process. However, if the iterative linear solver does not support scaling matrices (i.e., the `SUNLinSolSetScalingVectors` routine is `NULL`), then KINLS will be unable to fully handle ill-conditioning in the nonlinear solve process through the solution and residual scaling operators described in §2. In this case, KINLS will attempt to adjust the linear solver tolerance to account for this lack of functionality. To this end, the following assumptions are made:

1. All residual components have similar magnitude; hence the scaling matrix D_F used in computing the linear residual norm (see §2) should satisfy the assumption

$$(D_F)_{i,i} \approx D_{F,mean}, \quad \text{for } i = 0, \dots, n-1.$$

2. The SUNLINSOL object uses a standard 2-norm to measure convergence.

Since KINSOL uses D_F as the left-scaling matrix, $S_1 = D_F$, then the linear solver convergence requirement is converted as follows (using the notation from equations (9.1)-(9.2)):

$$\begin{aligned} & \|\tilde{b} - \tilde{A}\tilde{x}\|_2 < \text{tol} \\ \Leftrightarrow & \|D_F P_1^{-1} b - D_F P_1^{-1} A x\|_2 < \text{tol} \\ \Leftrightarrow & \sum_{i=0}^{n-1} [(D_F)_{i,i} (P_1^{-1}(b - Ax))_i]^2 < \text{tol}^2 \\ \Leftrightarrow & D_{F,mean}^2 \sum_{i=0}^{n-1} [(P_1^{-1}(b - Ax))_i]^2 < \text{tol}^2 \\ \Leftrightarrow & \sum_{i=0}^{n-1} [(P_1^{-1}(b - Ax))_i]^2 < \left(\frac{\text{tol}}{D_{F,mean}}\right)^2 \\ \Leftrightarrow & \|P_1^{-1}(b - Ax)\|_2 < \frac{\text{tol}}{D_{F,mean}} \end{aligned}$$

Therefore the tolerance scaling factor

$$D_{F,mean} = \frac{1}{\sqrt{n}} \left(\sum_{i=0}^{n-1} (D_F)_{i,i}^2 \right)^{1/2}$$

is computed and the scaled tolerance `delta` = `tol`/ $D_{F,mean}$ is supplied to the SUNLINSOL object.

9.4.3 Matrix-embedded solver incompatibility

At present, KINLS is incompatible with SUNLINSOL objects that self-identify as having type `SUNLINEARSOLVER_MATRIX_EMBEDDED`. Support for such user-supplied linear solvers may be added in a future release. Users interested in such support are recommended to contact the SUNDIALS development team.

9.5 The SUNLinearSolver_Dense implementation

This section describes the SUNLINSOL implementation for solving dense linear systems. The `SUNLINSOL_DENSE` module is designed to be used with the corresponding `SUNMATRIX_DENSE` matrix type, and one of the serial or shared-memory `NVECTOR` implementations (`NVECTOR_SERIAL`, `NVECTOR_OPENMP`, or `NVECTOR_PTHREADS`).

To access the `SUNLINSOL_DENSE` module, include the header file `sunlinsol/sunlinsol_dense.h`. We note that the `SUNLINSOL_DENSE` module is accessible from SUNDIALS packages *without* separately linking to the `libsundials_sunlinsoldense` module library.

9.5.1 SUNLinearSolver_Dense description

This solver is constructed to perform the following operations:

- The “setup” call performs a LU factorization with partial (row) pivoting ($\mathcal{O}(N^3)$ cost), $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix with 1’s on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX_DENSE object A , with pivoting information encoding P stored in the `pivots` array.
- The “solve” call performs pivoting and forward and backward substitution using the stored `pivots` array and the LU factors held in the SUNMATRIX_DENSE object ($\mathcal{O}(N^2)$ cost).

9.5.2 SUNLinearSolver_Dense functions

The SUNLINSOL_DENSE module provides the following user-callable constructor for creating a SUNLinearSolver object.

<code>SUNLinSol_Dense</code>	
Call	<code>LS = SUNLinSol_Dense(y, A);</code>
Description	The function <code>SUNLinSol_Dense</code> creates and allocates memory for a dense SUNLinearSolver object.
Arguments	<code>y</code> (N_Vector) a template for cloning vectors needed within the solver <code>A</code> (SUNMatrix) a SUNMATRIX_DENSE matrix template for cloning matrices needed within the solver
Return value	This returns a SUNLinearSolver object. If either <code>A</code> or <code>y</code> 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.
Deprecated Name	For backward compatibility, the wrapper function <code>SUNDenseLinearSolver</code> with identical input and output arguments is also provided.
F2003 Name	<code>FSUNLinSol_Dense</code>

The SUNLINSOL_DENSE module defines implementations of all “direct” linear solver operations listed in Sections 9.1.1 – 9.1.3:

- `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`

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’ to the function name.

9.5.3 SUNLinearSolver_Dense Fortran interfaces

The SUNLINSOL_DENSE module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The `fsunlinsol_dense_mod` FORTRAN module defines interfaces to all SUNLINSOL_DENSE C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function `SUNLinSol_Dense` is interfaced as `FSUNLinSol_Dense`.

The FORTRAN 2003 SUNLINSOL_DENSE interface module can be accessed with the `use` statement, i.e. `use fsunlinsol_dense_mod`, and linking to the library `libsundials_fsunlinsoldense_mod.lib` in addition to the C library. For details on where the library and module file `fsunlinsol_dense_mod.mod` are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators *without* separately linking to the `libsundials_fsunlinsoldense_mod` library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL_DENSE module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

FSUNDENSELINSOLINIT

Call	<code>FSUNDENSELINSOLINIT(code, ier)</code>
Description	The function <code>FSUNDENSELINSOLINIT</code> can be called for Fortran programs to create a dense <code>SUNLinearSolver</code> object.
Arguments	<code>code</code> (<code>int*</code>) is an integer input specifying the solver id (1 for <code>CVODE</code> , 2 for <code>IDA</code> , 3 for <code>KINSOL</code> , and 4 for <code>ARKODE</code>).
Return value	<code>ier</code> 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 <i>after</i> both the <code>NVECTOR</code> and <code>SUNMATRIX</code> 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	<code>FSUNMASSDENSELINSOLINIT(ier)</code>
Description	The function <code>FSUNMASSDENSELINSOLINIT</code> can be called for Fortran programs to create a dense <code>SUNLinearSolver</code> object for mass matrix linear systems.
Arguments	None
Return value	<code>ier</code> is a <code>int</code> 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 <i>after</i> both the <code>NVECTOR</code> and <code>SUNMATRIX</code> mass-matrix objects have been initialized.

9.5.4 SUNLinearSolver_Dense content

The SUNLINSOL_DENSE module defines the *content* field of a `SUNLinearSolver` as the following structure:

```

struct _SUNLinearSolverContent_Dense {
    sunindextype N;
    sunindextype *pivots;
    sunindextype 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.

9.6 The SUNLinearSolver_Band implementation

This section describes the SUNLINSOL implementation for solving banded linear systems. The SUNLINSOL_BAND module 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).

To access the SUNLINSOL_BAND module, include the header file `sunlinsol/sunlinsol_band.h`. We note that the SUNLINSOL_BAND module is accessible from SUNDIALS packages *without* separately linking to the `libsundials_sunlinsolband` module library.

9.6.1 SUNLinearSolver_Band description

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 `m1`, then the upper triangular factor U can have upper bandwidth as big as `smu = MIN(N-1, mu+m1)`. The lower triangular factor L has lower bandwidth `m1`.



9.6.2 SUNLinearSolver_Band functions

The SUNLINSOL_BAND module provides the following user-callable constructor for creating a SUNLinearSolver object.

<code>SUNLinSol_Band</code>	
Call	<code>LS = SUNLinSol_Band(y, A);</code>
Description	The function <code>SUNLinSol_Band</code> creates and allocates memory for a band <code>SUNLinearSolver</code> object.
Arguments	<code>y</code> (<code>N_Vector</code>) a template for cloning vectors needed within the solver <code>A</code> (<code>SUNMatrix</code>) a <code>SUNMATRIX_BAND</code> matrix template for cloning matrices needed within the solver
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>A</code> or <code>y</code> are incompatible then this routine will return <code>NULL</code> .

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_BAND matrix type and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

Additionally, this routine will verify that the input matrix *A* is allocated with appropriate upper bandwidth storage for the *LU* factorization.

Deprecated Name For backward compatibility, the wrapper function `SUNBandLinearSolver` with identical input and output arguments is also provided.

F2003 Name `FSUNLinSol_Band`

The `SUNLINSOL_BAND` module defines band implementations of all “direct” linear solver operations listed in Sections 9.1.1 – 9.1.3:

- `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`

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’ to the function name.

9.6.3 SUNLinearSolver_Band Fortran interfaces

The `SUNLINSOL_BAND` module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The `fsunlinsol_band_mod` FORTRAN module defines interfaces to all `SUNLINSOL_BAND` C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function `SUNLinSol_Band` is interfaced as `FSUNLinSol_Band`.

The FORTRAN 2003 `SUNLINSOL_BAND` interface module can be accessed with the `use` statement, i.e. `use fsunlinsol_band_mod`, and linking to the library `libsundials_fsunlinsolband_mod.lib` in addition to the C library. For details on where the library and module file `fsunlinsol_band_mod.mod` are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators *without* separately linking to the `libsundials_fsunlinsolband_mod` library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the `SUNLINSOL_BAND` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

FSUNBANDLINSOLINIT

Call	FSUNBANDLINSOLINIT(<i>code</i> , <i>ier</i>)
Description	The function FSUNBANDLINSOLINIT can be called for Fortran programs to create a band SUNLinearSolver object.
Arguments	<i>code</i> (<i>int*</i>) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
Return value	<i>ier</i> 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 <i>after</i> 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.

FSUNMASSBANDLINSOLINIT

Call	FSUNMASSBANDLINSOLINIT(<i>ier</i>)
Description	The function FSUNMASSBANDLINSOLINIT can be called for Fortran programs to create a band SUNLinearSolver object for mass matrix linear systems.
Arguments	None
Return value	<i>ier</i> is a <i>int</i> 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 <i>after</i> both the NVECTOR and SUNMATRIX mass-matrix objects have been initialized.

9.6.4 SUNLinearSolver_Band content

The SUNLINSOL_BAND module defines the *content* field of a SUNLinearSolver as the following structure:

```
struct _SUNLinearSolverContent_Band {
    sunindextype N;
    sunindextype *pivots;
    sunindextype 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.

9.7 The SUNLinearSolver_LapackDense implementation

This section describes the SUNLINSOL implementation for solving dense linear systems with LAPACK. The SUNLINSOL_LAPACKDENSE module is designed to be used with the corresponding SUNMATRIX_DENSE matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, or NVECTOR_PTHREADS).

To access the SUNLINSOL_LAPACKDENSE module, include the header file `sunlinsol/sunlinsol_lapackdense.h`. The installed module library to link to is `libsundials_sunlinsollapackdense.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The SUNLINSOL_LAPACKDENSE module 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 64-bit integers for the `sunindextype`.



9.7.1 SUNLinearSolver_LapackDense description

This solver is constructed to perform the following operations:

- The “setup” call performs a LU factorization with partial (row) pivoting ($\mathcal{O}(N^3)$ cost), $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix with 1’s on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input `SUNMATRIX_DENSE` object A , with pivoting information encoding P stored in the `pivots` array.
- The “solve” call performs pivoting and forward and backward substitution using the stored `pivots` array and the LU factors held in the `SUNMATRIX_DENSE` object ($\mathcal{O}(N^2)$ cost).

9.7.2 SUNLinearSolver_LapackDense functions

The SUNLINSOL_LAPACKDENSE module provides the following user-callable constructor for creating a `SUNLinearSolver` object.

`SUNLinSol_LapackDense`

Call	<code>LS = SUNLinSol_LapackDense(y, A);</code>
Description	The function <code>SUNLinSol_LapackDense</code> creates and allocates memory for a LAPACK-based, dense <code>SUNLinearSolver</code> object.
Arguments	<code>y</code> (<code>N_Vector</code>) a template for cloning vectors needed within the solver <code>A</code> (<code>SUNMatrix</code>) a <code>SUNMATRIX_DENSE</code> matrix template for cloning matrices needed within the solver
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>A</code> or <code>y</code> are incompatible then this routine will return <code>NULL</code> .
Notes	This routine will perform consistency checks to ensure that it is called with consistent <code>NVECTOR</code> and <code>SUNMATRIX</code> implementations. These are currently limited to the <code>SUNMATRIX_DENSE</code> matrix type and the <code>NVECTOR_SERIAL</code> , <code>NVECTOR_OPENMP</code> , and <code>NVECTOR_PTHREADS</code> vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.
Deprecated Name	For backward compatibility, the wrapper function <code>SUNLapackDense</code> with identical input and output arguments is also provided.

The SUNLINSOL_LAPACKDENSE module defines dense implementations of all “direct” linear solver operations listed in Sections 9.1.1 – 9.1.3:

- `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`

9.7.3 SUNLinearSolver_LapackDense Fortran interfaces

For solvers that include a FORTRAN 77 interface module, the `SUNLINSOL_LAPACKDENSE` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

FSUNLAPACKDENSEINIT

Call	<code>FSUNLAPACKDENSEINIT(code, ier)</code>
Description	The function <code>FSUNLAPACKDENSEINIT</code> can be called for Fortran programs to create a LAPACK-based dense <code>SUNLinearSolver</code> object.
Arguments	<code>code</code> (<code>int*</code>) is an integer input specifying the solver id (1 for <code>CVODE</code> , 2 for <code>IDA</code> , 3 for <code>KINSOL</code> , and 4 for <code>ARKODE</code>).
Return value	<code>ier</code> 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 <i>after</i> both the <code>NVECTOR</code> and <code>SUNMATRIX</code> 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.

FSUNMASSLAPACKDENSEINIT

Call	<code>FSUNMASSLAPACKDENSEINIT(ier)</code>
Description	The function <code>FSUNMASSLAPACKDENSEINIT</code> can be called for Fortran programs to create a LAPACK-based, dense <code>SUNLinearSolver</code> object for mass matrix linear systems.
Arguments	None
Return value	<code>ier</code> is a <code>int</code> 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 <i>after</i> both the <code>NVECTOR</code> and <code>SUNMATRIX</code> mass-matrix objects have been initialized.

9.7.4 SUNLinearSolver_LapackDense content

The `SUNLINSOL_LAPACKDENSE` module defines the *content* field of a `SUNLinearSolver` as the following structure:

```
struct _SUNLinearSolverContent_Dense {
    sunindextype N;
    sunindextype *pivots;
    sunindextype 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.

9.8 The SUNLinearSolver_LapackBand implementation

This section describes the SUNLINSOL implementation for solving banded linear systems with LAPACK. The SUNLINSOL_LAPACKBAND module 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).

To access the SUNLINSOL_LAPACKBAND module, include the header file `sunlinsol/sunlinsol_lapackband.h`. The installed module library to link to is `libsundials_sunlinsollapackband.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The SUNLINSOL_LAPACKBAND module 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 64-bit integers for the `sunindextype`.



9.8.1 SUNLinearSolver_LapackBand description

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 `m1`, then the upper triangular factor *U* can have upper bandwidth as big as `smu = MIN(N-1, mu+m1)`. The lower triangular factor *L* has lower bandwidth `m1`.



9.8.2 SUNLinearSolver_LapackBand functions

The SUNLINSOL_LAPACKBAND module provides the following user-callable constructor for creating a SUNLinearSolver object.

SUNLinSol_LapackBand

Call	<code>LS = SUNLinSol_LapackBand(y, A);</code>
Description	The function <code>SUNLinSol_LapackBand</code> creates and allocates memory for a LAPACK-based, band <code>SUNLinearSolver</code> object.
Arguments	<code>y</code> (<code>N_Vector</code>) 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_BAND matrix type and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check. Additionally, this routine will verify that the input matrix A is allocated with appropriate upper bandwidth storage for the LU factorization.
Deprecated Name	For backward compatibility, the wrapper function SUNLapackBand with identical input and output arguments is also provided.

The SUNLINSOL_LAPACKBAND module defines band implementations of all “direct” linear solver operations listed in Sections 9.1.1 – 9.1.3:

- **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**

9.8.3 SUNLinearSolver_LapackBand Fortran interfaces

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL_LAPACKBAND module also includes a Fortran-callable function for creating a **SUNLinearSolver** object.

FSUNLAPACKDENSEINIT

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 <i>after</i> 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 None

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.

9.8.4 SUNLinearSolver_LapackBand content

The SUNLINSOL_LAPACKBAND module defines the *content* field of a SUNLinearSolver as the following structure:

```
struct _SUNLinearSolverContent_Band {
    sunindextype N;
    sunindextype *pivots;
    sunindextype 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.

9.9 The SUNLinearSolver_KLU implementation

This section describes the SUNLINSOL implementation for solving sparse linear systems with KLU. The SUNLINSOL_KLU module is designed to be used with the corresponding SUNMATRIX_SPARSE matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, or NVECTOR_PTHREADS).

The header file to include when using this module is `sunlinsol/sunlinsol_klu.h`. The installed module library to link to is `libsundials_sunlinsolklu.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The SUNLINSOL_KLU module is a SUNLINSOL wrapper for the KLU sparse matrix factorization and solver library written by Tim Davis [3, 16]. In order to use the SUNLINSOL_KLU interface to KLU, it is assumed that KLU has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with KLU (see Appendix A for details). Additionally, this wrapper only supports double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to have `realtype` set to either `extended` or `single` (see Section 4.2). Since the KLU library supports both 32-bit and 64-bit integers, this interface will be compiled for either of the available `sunindextype` options.



9.9.1 SUNLinearSolver_KLU description

The KLU library has a symbolic factorization routine that computes the permutation of the linear system matrix to block triangular form and the permutations that will pre-order the diagonal blocks (the only ones that need to be factored) to reduce fill-in (using AMD, COLAMD, CHOLAMD, natural, or an ordering given by the user). Of these ordering choices, the default value in the SUNLINSOL_KLU module is the COLAMD ordering.

KLU breaks the factorization into two separate parts. The first is a symbolic factorization and the second is a numeric factorization that returns the factored matrix along with final pivot information. KLU also has a refactor routine that can be called instead of the numeric factorization. This routine will reuse the pivot information. This routine also returns diagnostic information that a user can examine to determine if numerical stability is being lost and a full numerical factorization should be done instead of the refactor.

Since the linear systems that arise within the context of SUNDIALS calculations will typically have identical sparsity patterns, the SUNLINSOL_KLU module is constructed to perform the following operations:

- The first time that the “setup” routine is called, it performs the symbolic factorization, followed by an initial numerical factorization.
- On subsequent calls to the “setup” routine, it calls the appropriate KLU “refactor” routine, followed by estimates of the numerical conditioning using the relevant “rcond”, and if necessary “condest”, routine(s). If these estimates of the condition number are larger than $\varepsilon^{-2/3}$ (where ε is the double-precision unit roundoff), then a new factorization is performed.
- The module includes the routine `SUNKLUReInit`, that can be called by the user to force a full 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.

9.9.2 SUNLinearSolver_KLU functions

The SUNLINSOL_KLU module provides the following user-callable constructor for creating a `SUNLinearSolver` object.

<code>SUNLinSol_KLU</code>	
Call	<code>LS = SUNLinSol_KLU(y, A);</code>
Description	The function <code>SUNLinSol_KLU</code> creates and allocates memory for a KLU-based <code>SUNLinearSolver</code> object.
Arguments	<code>y</code> (<code>N_Vector</code>) a template for cloning vectors needed within the solver <code>A</code> (<code>SUNMatrix</code>) a <code>SUNMATRIX_SPARSE</code> matrix template for cloning matrices needed within the solver
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>A</code> or <code>y</code> are incompatible then this routine will return <code>NULL</code> .
Notes	This routine will perform consistency checks to ensure that it is called with consistent <code>NVECTOR</code> and <code>SUNMATRIX</code> implementations. These are currently limited to the <code>SUNMATRIX_SPARSE</code> matrix type (using either CSR or CSC storage formats) and the <code>NVECTOR_SERIAL</code> , <code>NVECTOR_OPENMP</code> , and <code>NVECTOR_PTHREADS</code> vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.
Deprecated Name	For backward compatibility, the wrapper function <code>SUNKLU</code> with identical input and output arguments is also provided.
F2003 Name	<code>FSUNLinSol_KLU</code>

The SUNLINSOL_KLU module defines implementations of all “direct” linear solver operations listed in Sections 9.1.1 – 9.1.3:

- `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`

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’ to the function name.

The `SUNLINSOL_KLU` module also defines the following additional user-callable functions.

`SUNLinSol_KLUReInit`

Call	<code>retval = SUNLinSol_KLUReInit(LS, A, nnz, reinit_type);</code>
Description	The function <code>SUNLinSol_KLUReInit</code> 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	<p><code>LS</code> (SUNLinearSolver) a template for cloning vectors needed within the solver</p> <p><code>A</code> (SUNMatrix) a <code>SUNMATRIX_SPARSE</code> matrix template for cloning matrices needed within the solver</p> <p><code>nnz</code> (sunindextype) the new number of nonzeros in the matrix</p> <p><code>reinit_type</code> (int) flag governing the level of reinitialization. The allowed values are:</p> <ul style="list-style-type: none"> • <code>SUNKLU_REINIT_FULL</code> – The Jacobian matrix will be destroyed and a new one will be allocated based on the <code>nnz</code> value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup. • <code>SUNKLU_REINIT_PARTIAL</code> – Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of <code>nnz</code> given in the sparse matrix provided to the original constructor routine (or the previous <code>SUNLinSol_KLUReInit</code> call).
Return value	The return values from this function are <code>SUNLS_MEM_NULL</code> (either <code>S</code> or <code>A</code> are <code>NULL</code>), <code>SUNLS_ILL_INPUT</code> (<code>A</code> does not have type <code>SUNMATRIX_SPARSE</code> or <code>reinit_type</code> is invalid), <code>SUNLS_MEM_FAIL</code> (reallocation of the sparse matrix failed) or <code>SUNLS_SUCCESS</code> .
Notes	<p>This routine will perform consistency checks to ensure that it is called with consistent <code>NVECTOR</code> and <code>SUNMATRIX</code> implementations. These are currently limited to the <code>SUNMATRIX_SPARSE</code> matrix type (using either <code>CSR</code> or <code>CSC</code> storage formats) and the <code>NVECTOR_SERIAL</code>, <code>NVECTOR_OPENMP</code>, and <code>NVECTOR_PTHREADS</code> vector types. As additional compatible matrix and vector implementations are added to <code>SUNDIALS</code>, these will be included within this compatibility check.</p> <p>This routine assumes no other changes to solver use are necessary.</p>

Deprecated Name For backward compatibility, the wrapper function `SUNKLUReInit` with identical input and output arguments is also provided.

F2003 Name `FSUNLinSol_KLUReInit`

`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 indicating the reordering algorithm to use, the options are:

- 0 AMD,
- 1 COLAMD, and
- 2 the natural ordering.

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 choice), or `SUNLS_SUCCESS`.

Deprecated Name For backward compatibility, the wrapper function `SUNKLUSetOrdering` with identical input and output arguments is also provided.

F2003 Name `FSUNLinSol_KLUSetOrdering`

`SUNLinSol_KLUGetSymbolic`

Call `symbolic = SUNLinSol_KLUGetSymbolic(LS);`

Description This function returns a pointer to the KLU symbolic factorization stored in the `SUNLINSOL_KLU content` structure.

Arguments `LS` (`SUNLinearSolver`) the `SUNLINSOL_KLU` object

Return value The return type from this function is `sun_klu_symbolic`.

Notes When SUNDIALS is compiled with 32-bit indices (`SUNDIALS_INDEX_SIZE=32`), `sun_klu_symbolic` is mapped to the KLU type `klu_symbolic`; when SUNDIALS is compiled with 64-bit indices (`SUNDIALS_INDEX_SIZE=64`) this is mapped to the KLU type `klu_l_symbolic`.

`SUNLinSol_KLUGetNumeric`

Call `numeric = SUNLinSol_KLUGetNumeric(LS);`

Description This function returns a pointer to the KLU numeric factorization stored in the `SUNLINSOL_KLU content` structure.

Arguments `LS` (`SUNLinearSolver`) the `SUNLINSOL_KLU` object

Return value The return type from this function is `sun_klu_numeric`.

Notes When SUNDIALS is compiled with 32-bit indices (`SUNDIALS_INDEX_SIZE=32`), `sun_klu_numeric` is mapped to the KLU type `klu_numeric`; when SUNDIALS is compiled with 64-bit indices (`SUNDIALS_INDEX_SIZE=64`), this is mapped to the KLU type `klu_l_numeric`.

SUNLinSol_KLUGetCommon

Call	<code>common = SUNLinSol_KLUGetCommon(LS);</code>
Description	This function returns a pointer to the KLU common structure stored within in the SUNLINSOL_KLU content structure.
Arguments	LS (SUNLinearSolver) the SUNLINSOL_KLU object
Return value	The return type from this function is <code>sun_klu_common</code> .
Notes	When SUNDIALS is compiled with 32-bit indices (SUNDIALS_INDEX_SIZE=32), <code>sun_klu_common</code> is mapped to the KLU type <code>klu_common</code> ; when SUNDIALS is compiled with 64-bit indices (SUNDIALS_INDEX_SIZE=64), this is mapped to the KLU type <code>klu_l_common</code> .

9.9.3 SUNLinearSolver_KLU Fortran interfaces

The SUNLINSOL_KLU module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The `fsunlinsol_klu_mod` FORTRAN module defines interfaces to all SUNLINSOL_KLU C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function `SUNLinSol_klu` is interfaced as `FSUNLinSol_klu`.

The FORTRAN 2003 SUNLINSOL_KLU interface module can be accessed with the `use` statement, i.e. `use fsunlinsol_klu_mod`, and linking to the library `libsundials_fsunlinsolklu_mod.lib` in addition to the C library. For details on where the library and module file `fsunlinsol_klu_mod.mod` are installed see [Appendix A](#).

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL_KLU module also includes a Fortran-callable function for creating a SUNLinearSolver object.

FSUNKLUINIT

Call	<code>FSUNKLUINIT(code, ier)</code>
Description	The function FSUNKLUINIT can be called for Fortran programs to create a SUNLINSOL_KLU object.
Arguments	<code>code</code> (<code>int*</code>) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
Return value	<code>ier</code> 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 <i>after</i> both the NVECTOR and SUNMATRIX objects have been initialized.

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

FSUNMASSKLUINIT

Call	<code>FSUNMASSKLUINIT(ier)</code>
Description	The function FSUNMASSKLUINIT can be called for Fortran programs to create a KLU-based SUNLinearSolver object for mass matrix linear systems.

Arguments None

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.

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 `SUNLINSOL_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`).

`nnz` (`sunindextype*`) the new number of nonzeros in the matrix

`reinit_type` (`int*`) flag governing the level of reinitialization. The allowed values are:

- 1 – The Jacobian matrix will be destroyed and a new one will be allocated based on the `nnz` value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup.
- 2 – Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of `nnz` given in the sparse matrix provided to the original constructor routine (or the previous `SUNLinSol_KLUReInit` call).

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

`ordering` (`int*`) flag indication the reordering algorithm to use. Options include:

- 0 AMD,
- 1 COLAMD, and

2 the natural ordering.

The default is 1 for COLAMD.

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.

FSUNMASSKLUSETORDERING

Call `FSUNMASSKLUSETORDERING(ier)`

Description The function `FSUNMASSKLUSETORDERING` 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 `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.

9.9.4 SUNLinearSolver_KLU content

The `SUNLINSOL_KLU` module defines the *content* field of a `SUNLinearSolver` as the following structure:

```
struct _SUNLinearSolverContent_KLU {
    int          last_flag;
    int          first_factorize;
    sun_klu_symbolic *symbolic;
    sun_klu_numeric *numeric;
    sun_klu_common common;
    sunindextype (*klu_solver)(sun_klu_symbolic*, sun_klu_numeric*,
                               sunindextype, sunindextype,
                               double*, sun_klu_common*);
};
```

These entries of the *content* field contain the following information:

`last_flag` - last error return flag from internal function evaluations,

`first_factorize` - flag indicating whether the factorization has ever been performed,

`symbolic` - KLU storage structure for symbolic factorization components, with underlying type `klu_symbolic` or `klu_l_symbolic`, depending on whether SUNDIALS was installed with 32-bit versus 64-bit indices, respectively,

`numeric` - KLU storage structure for numeric factorization components, with underlying type `klu_numeric` or `klu_l_numeric`, depending on whether SUNDIALS was installed with 32-bit versus 64-bit indices, respectively.

`common` - storage structure for common KLU solver components, with underlying type `klu_common` or `klu_l_common`, depending on whether SUNDIALS was installed with 32-bit versus 64-bit indices, respectively,

`klu_solver` - pointer to the appropriate KLU solver function (depending on whether it is using a CSR or CSC sparse matrix, and on whether SUNDIALS was installed with 32-bit or 64-bit indices).

9.10 The SUNLinearSolver_SuperLUDIST implementation

The SuperLU_DIST implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_SUPERLUDIST, is designed to be used with the corresponding SUNMATRIX_SLUNRLOC matrix type, and one of the serial, threaded or parallel NVECTOR implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, NVECTOR_PTHREADS, NVECTOR_PARALLEL, or NVECTOR_PARHYP).

The header file to include when using this module is `sunlinsol/sunlinsol_superludist.h`. The installed module library to link to is `libsundials_sunlinsol_superludist.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

9.10.1 SUNLinearSolver_SuperLUDIST description

The SUNLINSOL_SUPERLUDIST module is a SUNLINSOL adapter for the SuperLU_DIST sparse matrix factorization and solver library written by X. Sherry Li [8, 24, 32, 33]. The package uses a SPMD parallel programming model and multithreading to enhance efficiency in distributed-memory parallel environments with multicore nodes and possibly GPU accelerators. It uses MPI for communication, OpenMP for threading, and CUDA for GPU support. In order to use the SUNLINSOL_SUPERLUDIST interface to SuperLU_DIST, it is assumed that SuperLU_DIST has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with SuperLU_DIST (see Appendix A for details). Additionally, the adapter only supports double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to use single or extended precision. Moreover, since the SuperLU_DIST library may be installed to support either 32-bit or 64-bit integers, it is assumed that the SuperLU_DIST library is installed using the same integer size as SUNDIALS.

The SuperLU_DIST library provides many options to control how a linear system will be solved. These options may be set by a user on an instance of the `superlu_dist_options_t` struct, and then it may be provided as an argument to the SUNLINSOL_SUPERLUDIST constructor. The SUNLINSOL_SUPERLUDIST module will respect all options set except for `Fact` – this option is necessarily modified by the SUNLINSOL_SUPERLUDIST module in the setup and solve routines.

Since the linear systems that arise within the context of SUNDIALS calculations will typically have identical sparsity patterns, the SUNLINSOL_SUPERLUDIST module is constructed to perform the following operations:

- The first time that the “setup” routine is called, it sets the SuperLU_DIST option `Fact` to `DOFACT` so that a subsequent call to the “solve” routine will perform a symbolic factorization, followed by an initial numerical factorization before continuing to solve the system.
- On subsequent calls to the “setup” routine, it sets the SuperLU_DIST option `Fact` to `SamePattern` so that a subsequent call to “solve” will perform factorization assuming the same sparsity pattern as prior, i.e. it will reuse the column permutation vector.
- If “setup” is called prior to the “solve” routine, then the “solve” routine will perform a symbolic factorization, followed by an initial numerical factorization before continuing to the sparse triangular solves, and, potentially, iterative refinement. If “setup” is not called prior, “solve” will skip to the triangular solve step. We note that in this solve SuperLU_DIST operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

Starting with SuperLU_DIST version 6.3.0, some structures were renamed to have a prefix for the floating point type. The double precision API functions have the prefix `'d'`. To maintain backwards compatibility with the unprefix types, SUNDIALS provides macros to these SuperLU_DIST types with an `'x'` prefix that expand to the correct prefix. E.g., the SUNDIALS macro `xLUstruct_t` expands to `dLUstruct_t` or `LUstruct_t` based on the SuperLU_DIST version.



9.10.2 SUNLinearSolver_SuperLUDIST functions

The SUNLINSOL_SUPERLUDIST module defines implementations of all “direct” linear solver operations listed in Sections 9.1.1-9.1.3:

- `SUNLinSolGetType_SuperLUDIST`
- `SUNLinSolInitialize_SuperLUDIST` – this sets the `first_factorize` flag to 1 and resets the internal SuperLU_DIST statistics variables.
- `SUNLinSolSetup_SuperLUDIST` – this sets the appropriate SuperLU_DIST options so that a subsequent solve will perform a symbolic and numerical factorization before proceeding with the triangular solves
- `SUNLinSolSolve_SuperLUDIST` – this calls the SuperLU_DIST solve routine to perform factorization (if the setup routine was called prior) and then use the *LU* factors to solve the linear system.
- `SUNLinSolLastFlag_SuperLUDIST`
- `SUNLinSolSpace_SuperLUDIST` – 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 SuperLU_DIST documentation.
- `SUNLinSolFree_SuperLUDIST`

In addition, the module SUNLINSOL_SUPERLUDIST provides the following user-callable routines:

<code>SUNLinSol_SuperLUDIST</code>

Call `LS = SUNLinSol_SuperLUDIST(y, A, grid, lu, scaleperm, solve, stat, options);`

Description The function `SUNLinSol_SuperLUDIST` creates and allocates memory for a `SUNLINSOL_SUPERLUDIST` object.

Arguments

<code>y</code>	(<code>N_Vector</code>) a template for cloning vectors needed within the solver
<code>A</code>	(<code>SUNMatrix</code>) a <code>SUNMATRIX_SLUNRLOC</code> matrix template for cloning matrices needed within the solver
<code>grid</code>	(<code>gridinfo_t*</code>)
<code>lu</code>	(<code>LUstruct_t*</code>)
<code>scaleperm</code>	(<code>ScalePermstruct_t*</code>)
<code>solve</code>	(<code>SOLVEstruct_t*</code>)
<code>stat</code>	(<code>SuperLUStat_t*</code>)
<code>options</code>	(<code>superlu_dist_options_t*</code>)

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 SuperLU_DIST 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_SLUNRLOC` matrix type and the `NVECTOR_SERIAL`, `NVECTOR_PARALLEL`, `NVECTOR_PARHYP`, `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 `grid`, `lu`, `scaleperm`, `solve`, and `options` arguments are not checked and are passed directly to SuperLU_DIST routines.

Some struct members of the `options` argument are modified internally by the `SUNLINSOL_SUPERLUDIST` solver. Specifically the member `Fact`, is modified in the setup and solve routines.

SUNLinSol_SuperLUDIST_GetBerr

Call `realtype berr = SUNLinSol_SuperLUDIST_GetBerr(LS);`

Description The function `SUNLinSol_SuperLUDIST_GetBerr` returns the componentwise relative backward error of the computed solution.

Arguments `LS` (`SUNLinearSolver`) the `SUNLINSOL_SUPERLUDIST` object

Return value `realtype`

Notes

SUNLinSol_SuperLUDIST_GetGridinfo

Call `gridinfo_t *grid = SUNLinSol_SuperLUDIST_GetGridinfo(LS);`

Description The function `SUNLinSol_SuperLUDIST_GetGridinfo` returns the `SuperLU_DIST` structure that contains the 2D process grid.

Arguments `LS` (`SUNLinearSolver`) the `SUNLINSOL_SUPERLUDIST` object

Return value `gridinfo_t*`

Notes

SUNLinSol_SuperLUDIST_GetLUstruct

Call `LUstruct_t *lu = SUNLinSol_SuperLUDIST_GetLUstruct(LS);`

Description The function `SUNLinSol_SuperLUDIST_GetLUstruct` returns the `SuperLU_DIST` structure that contains the distributed L and U factors.

Arguments `LS` (`SUNLinearSolver`) the `SUNLINSOL_SUPERLUDIST` object

Return value `LUstruct_t*`

Notes

SUNLinSol_SuperLUDIST_GetSuperLUOptions

Call `superlu_dist_options_t *opts = SUNLinSol_SuperLUDIST_GetSuperLUOptions(LS);`

Description The function `SUNLinSol_SuperLUDIST_GetSuperLUOptions` returns the `SuperLU_DIST` structure that contains the options which control how the linear system is factorized and solved.

Arguments `LS` (`SUNLinearSolver`) the `SUNLINSOL_SUPERLUDIST` object

Return value `superlu_dist_options_t*`

Notes

SUNLinSol_SuperLUDIST_GetScalePermstruct

Call `ScalePermstruct_t *sp = SUNLinSol_SuperLUDIST_GetScalePermstruct(LS);`

Description The function `SUNLinSol_SuperLUDIST_GetScalePermstruct` returns the `SuperLU_DIST` structure that contains the vectors that describe the transformations done to the matrix, A .

Arguments `LS` (`SUNLinearSolver`) the `SUNLINSOL_SUPERLUDIST` object

Return value `ScalePermstruct_t*`

Notes

SUNLinSol_SuperLUDIST_GetSOLVEstruct

Call `SOLVEstruct_t *solve = SUNLinSol_SuperLUDIST_GetSOLVEstruct (LS);`

Description The function `SUNLinSol_SuperLUDIST_GetSOLVEstruct` returns the `SuperLU_DIST` structure that contains information for communication during the solution phase.

Arguments `LS` (`SUNLinearSolver`) the `SUNLINSOL_SUPERLUDIST` object

Return value `SOLVEstruct_t*`

Notes

SUNLinSol_SuperLUDIST_GetSuperLUStat

Call `SuperLUStat_t *stat = SUNLinSol_SuperLUDIST_GetSuperLUStat (LS);`

Description The function `SUNLinSol_SuperLUDIST_GetSuperLUStat` returns the `SuperLU_DIST` structure that stores information about runtime and flop count.

Arguments `LS` (`SUNLinearSolver`) the `SUNLINSOL_SUPERLUDIST` object

Return value `SuperLUStat_t*`

Notes

9.10.3 SUNLinearSolver_SuperLUDIST content

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

```
struct _SUNLinearSolverContent_SuperLUDIST {
    booleantype    first_factorize;
    int           last_flag;
    reatype       berr;
    gridinfo_t    *grid;
    xLUstruct_t   *lu;
    superlu_dist_options_t *options;
    xScalePermstruct_t *scaleperm;
    xSOLVEstruct_t *solve;
    SuperLUStat_t *stat;
    sunindextype  N;
};
```

These entries of the *content* field contain the following information:

first_factorize - flag indicating whether the factorization has ever been performed,

last_flag - last error return flag from calls to internal routines,

berr - the componentwise relative backward error of the computed solution,

grid - pointer to the `SuperLU_DIST` structure that stores the 2D process grid,

lu - pointer to the `SuperLU_DIST` structure that stores the distributed L and U factors,

options - pointer to `SuperLU_DIST` options structure,

scaleperm - pointer to the `SuperLU_DIST` structure that stores vectors describing the transformations done to the matrix, A ,

solve - pointer to the `SuperLU_DIST` solve structure,

stat - pointer to the `SuperLU_DIST` structure that stores information about runtime and flop count,

N - the number of equations in the system

9.11 The SUNLinearSolver_SuperLUMT implementation

This section describes the SUNLINSOL implementation for solving sparse linear systems with SuperLU_MT. The SUPERLUMT module is designed to be used with the corresponding SUNMATRIX_SPARSE matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, or NVECTOR_PTHREADS). While these are compatible, it is not recommended to use a threaded vector module with SUNLINSOL_SUPERLUMT unless it is the NVECTOR_OPENMP module and the SUPERLUMT library has also been compiled with OpenMP.

The header file to include when using this module is `sunlinsol/sunlinsol_superlumt.h`. The installed module library to link to is `libsundials_sunlinsolsuperlumt.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The SUNLINSOL_SUPERLUMT module is a SUNLINSOL wrapper for the SUPERLUMT sparse matrix factorization and solver library written by X. Sherry Li [9, 31, 18]. 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.



9.11.1 SUNLinearSolver_SuperLUMT description

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.

9.11.2 SUNLinearSolver_SuperLUMT functions

The module SUNLINSOL_SUPERLUMT provides the following user-callable constructor for creating a `SUNLinearSolver` object.

<code>SUNLinSol_SuperLUMT</code>	
Call	<code>LS = SUNLinSol_SuperLUMT(y, A, num.threads);</code>
Description	The function <code>SUNLinSol_SuperLUMT</code> creates and allocates memory for a SuperLU_MT-based <code>SUNLinearSolver</code> object.
Arguments	<code>y</code> (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		<p>This routine analyzes the input matrix and vector to determine the linear system size and to assess compatibility with the SUPERLUMT library.</p> <p>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.</p> <p>The num_threads argument is not checked and is passed directly to SUPERLUMT routines.</p>
Deprecated Name		For backward compatibility, the wrapper function SUNSuperLUMT with identical input and output arguments is also provided.

The **SUNLINSOL_SUPERLUMT** module defines implementations of all “direct” linear solver operations listed in Sections 9.1.1 – 9.1.3:

- **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**

The **SUNLINSOL_SUPERLUMT** module also defines the following additional user-callable function.

SUNLinSol_SuperLUMTSetOrdering

Call	<code>retval = SUNLinSol_SuperLUMTSetOrdering(LS, ordering);</code>
Description	This function sets the ordering used by SUPERLUMT for reducing fill in the linear solve.
Arguments	<p>LS (SUNLinearSolver) the SUNLINSOL_SUPERLUMT object</p> <p>ordering (int) a flag indicating the ordering algorithm to use, the options are:</p> <ul style="list-style-type: none"> 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`.
- Deprecated Name For backward compatibility, the wrapper function `SUNSuperLUMTSetOrdering` with identical input and output arguments is also provided.

9.11.3 SUNLinearSolver_SuperLUMT Fortran interfaces

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

FSUNSUPERLUMTINIT

- Call `FSUNSUPERLUMTINIT(code, num_threads, ier)`
- Description The function `FSUNSUPERLUMTINIT` can be called for Fortran programs to create a `SUNLINSOL_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`).
`num_threads` (`int*`) desired number of threads (OpenMP or Pthreads, depending on how `SUPERLUMT` was installed) to use during the factorization steps
- 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_SUPERLUMT` module includes a Fortran-callable function for creating a `SUNLinearSolver` mass matrix solver object.

FSUNMASSSUPERLUMTINIT

- Call `FSUNMASSSUPERLUMTINIT(num_threads, ier)`
- Description The function `FSUNMASSSUPERLUMTINIT` can be called for Fortran programs to create a SuperLU_MT-based `SUNLinearSolver` object for mass matrix linear systems.
- Arguments `num_threads` (`int*`) desired number of threads (OpenMP or Pthreads, depending on how `SUPERLUMT` was installed) to use during the factorization steps.
- 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.

The `SUNLinSol_SuperLUMTSetOrdering` routine also supports Fortran interfaces for the system and mass matrix solvers:

FSUNSUPERLUMTSETORDERING

- Call `FSUNSUPERLUMTSETORDERING(code, ordering, ier)`
- Description The function `FSUNSUPERLUMTSETORDERING` can be called for Fortran programs to update the ordering algorithm in a `SUNLINSOL_SUPERLUMT` 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`).
`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 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_SuperLUMTSetOrdering` for complete further documentation of this routine.

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 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_SuperLUMTSetOrdering` for complete further documentation of this routine.

9.11.4 SUNLinearSolver_SuperLUMT content

The `SUNLINSOL_SUPERLUMT` module defines the *content* field of a `SUNLinearSolver` as the following structure:

```
struct _SUNLinearSolverContent_SuperLUMT {
    int      last_flag;
    int      first_factorize;
    SuperMatrix *A, *AC, *L, *U, *B;
    Gstat_t  *Gstat;
    sunindextype *perm_r, *perm_c;
    sunindextype N;
    int      num_threads;
    realtype  diag_pivot_thresh;
    int      ordering;
    superlumt_options_t *options;
};
```

These entries of the *content* field contain the following information:

- `last_flag` - last error return flag from internal function evaluations,
- `first_factorize` - flag indicating whether the factorization has ever been performed,
- `A, AC, L, U, B` - `SuperMatrix` pointers used in solve,
- `Gstat` - `GStat_t` object used in solve,
- `perm_r, perm_c` - permutation arrays used in solve,
- `N` - size of the linear system,

num_threads - number of OpenMP/Pthreads threads to use,
diag_pivot_thresh - threshold on diagonal pivoting,
ordering - flag for which reordering algorithm to use,
options - pointer to SUPERLUMT options structure.

9.12 The SUNLinearSolver_cuSolverSp_batchQR implementation

The SUNLinearSolver_cuSolverSp_batchQR implementation of the SUNLINSOL API is designed to be used with the SUNMATRIX_CUSPARSE matrix, and the NVECTOR_CUDA vector. The header file to include when using this module is `sunlinsol/sunlinsol_cusolversp_batchqr.h`. The installed library to link to is `libsundials_sunlinsolcusolversp.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The SUNLinearSolver_cuSolverSp_batchQR module is experimental and subject to change.



9.12.1 SUNLinearSolver_cuSolverSp_batchQR description

The SUNLinearSolver_cuSolverSp_batchQR implementation provides an interface to the batched sparse QR factorization method provided by the NVIDIA cuSOLVER library [6]. The module is designed for solving block diagonal linear systems of the form

$$\begin{bmatrix} \mathbf{A}_1 & 0 & \cdots & 0 \\ 0 & \mathbf{A}_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A}_n \end{bmatrix} x_j = b_j$$

where all block matrices \mathbf{A}_j share the same sparsity pattern. The matrix must be the SUNMATRIX_CUSPARSE module.

9.12.2 SUNLinearSolver_cuSolverSp_batchQR functions

The SUNLinearSolver_cuSolverSp_batchQR module defines implementations of all “direct” linear solver operations listed in Sections 9.1.1-9.1.3:

- `SUNLinSolGetType_cuSolverSp_batchQR`
- `SUNLinSolInitialize_cuSolverSp_batchQR` – this sets the `first_factorize` flag to 1
- `SUNLinSolSetup_cuSolverSp_batchQR` – this always copies the relevant SUNMATRIX_SPARSE data to the GPU; if this is the first setup it will perform symbolic analysis on the system
- `SUNLinSolSolve_cuSolverSp_batchQR` – this calls the `cusolverSpXcsrqrsvBatched` routine to perform factorization
- `SUNLinSolLastFlag_cuSolverSp_batchQR`
- `SUNLinSolFree_cuSolverSp_batchQR`

In addition, the module provides the following user-callable routines:

SUNLinSol_cuSolverSp_batchQR

Call `LS = SUNLinSol_cuSolverSp_batchQR(y, A, cusol);`

Description The function `SUNLinSol_cuSolverSp_batchQR` creates and allocates memory for a `SUNLINSOL` object.

Arguments `y` (`N_Vector`) a `NVECTOR_CUDA` vector for checking compatibility with the solver
`A` (`SUNMatrix`) a `SUNMATRIX_SPARSE` matrix for checking compatibility with the solver
`cusol` (`cusolverHandle_t`) a valid `cuSOLVER` handle

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 solver.

This routine will perform consistency checks to ensure that it is called with consistent `NVECTOR` and `SUNMATRIX` implementations. These are currently limited to the `SUNMAT_CUSPARSE` matrix type and the `NVECTOR_CUDA` vector type. As additional compatible matrix and vector implementations are added to `SUNDIALS`, these will be included within this compatibility check.

SUNLinSol_cuSolverSp_batchQR.GetDescription

Call `SUNLinSol_cuSolverSp_batchQR.GetDescription(LS, &desc);`

Description The function `SUNLinSol_cuSolverSp_batchQR.GetDescription` accesses the string description of the object (empty by default).

Arguments `LS` (`SUNLinearSolver`) a `SUNLinSol_cuSolverSp_batchQR` object
`desc` (`char **`) the string description of the linear solver

Return value None

SUNLinSol_cuSolverSp_batchQR.SetDescription

Call `SUNLinSol_cuSolverSp_batchQR.SetDescription(LS, desc);`

Description The function `SUNLinSol_cuSolverSp_batchQR.SetDescription` sets the string description of the object (empty by default).

Arguments `LS` (`SUNLinearSolver`) a `SUNLinSol_cuSolverSp_batchQR` object
`desc` (`const char *`) the string description of the linear solver

Return value None

SUNLinSol_cuSolverSp_batchQR.GetDeviceSpace

Call `SUNLinSol_cuSolverSp_batchQR.GetDeviceSpace(LS, cuSolverInternal, cuSolverWorkspace);`

Description The function `SUNLinSol_cuSolverSp_batchQR.GetDeviceSpace` returns the `cuSOLVER` batch QR method internal buffer size, in bytes, in the argument `cuSolverInternal` and the `cuSOLVER` batch QR workspace buffer size, in bytes, in the argument `cuSolverWorkspace`. The size of the internal buffer is proportional to the number of matrix blocks while the size of the workspace is almost independent of the number of blocks.

Arguments `LS` (`SUNLinearSolver`) a `SUNLinSol_cuSolverSp_batchQR` object
`cuSolverInternal` (`size_t *`) output – the size of the `cuSOLVER` internal buffer in bytes

cuSolverWorkspace (size_t *) output – the size of the cuSOLVER workspace buffer in bytes

Return value None

9.12.3 SUNLinearSolver_cuSolverSp_batchQR content

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

```
struct _SUNLinearSolverContent_cuSolverSp_batchQR {
    int          last_flag;          /* last return flag */
    booleantype  first_factorize;    /* is this the first factorization? */
    size_t       internal_size;      /* size of cusolver internal buffer for Q and R */
    size_t       workspace_size;     /* size of cusolver memory block for num. factorization */
    cusolverSpHandle_t  cusolver_handle; /* cuSolverSp context */
    csrqrInfo_t  info;              /* opaque cusolver data structure */
    void*        workspace;         /* memory block used by cusolver */
    const char*  desc;              /* description of this linear solver */
};
```

9.13 The SUNLinearSolver_MagmaDense implementation

The SUNLinearSolver_MagmaDense implementation of the SUNLINSOL API is designed to be used with the SUNMATRIX_MAGMADENSE matrix, and a GPU-enabled vector. This implementation interfaces to the MAGMA () linear algebra library and can target NVIDIA’s CUDA programming model or AMD’s HIP programming model [39].

The header file to include when using this module is sunlinsol/sunlinsol_magmadense.h. The installed library to link to is libsundials_sunlinsolmagmadense.lib where .lib is typically .so for shared libraries and .a for static libraries.

The SUNLinearSolver_MagmaDense module is experimental and subject to change.



9.13.1 SUNLinearSolver_MagmaDense description

The SUNLinearSolver_MagmaDense implementation provides an interface to the dense LU and dense batched LU methods in the MAGMA linear algebra library [4]. The batched LU methods are leveraged when solving block diagonal linear systems of the form

$$\begin{bmatrix} \mathbf{A}_0 & 0 & \cdots & 0 \\ 0 & \mathbf{A}_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A}_{n-1} \end{bmatrix} x_j = b_j.$$

9.13.2 SUNLinearSolver_MagmaDense functions

The SUNLinearSolver_MagmaDense module defines implementations of all “direct” linear solver operations listed in Sections 9.1.1-9.1.3:

- SUNLinSolGetType_MagmaDense
- SUNLinSolInitialize_MagmaDense
- SUNLinSolSetup_MagmaDense
- SUNLinSolSolve_MagmaDense

- SUNLinSolLastFlag_MagmaDense
- SUNLinSolFree_MagmaDense

In addition, the module provides the following user-callable routines:

SUNLinSol_MagmaDense

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

Description The function `SUNLinSol_MagmaDense` creates and allocates memory for a `SUNLINSOL` object.

Arguments `y` (`N_Vector`) a vector for checking compatibility with the solver
`A` (`SUNMatrix`) a `SUNMATRIX_MAGMADENSE` matrix for checking compatibility with 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 analyzes the input matrix and vector to determine the linear system size and to assess compatibility with the solver.

SUNLinSol_MagmaDense_SetAsync

Call `SUNLinSol_MagmaDense_SetAsync(SUNLinearSolver LS, booleantype onoff);`

Description The function `SUNLinSol_MagmaDense_SetAsync` can be used to toggle the linear solver between asynchronous and synchronous modes. In asynchronous mode, `SUNLinearSolver` operations are asynchronous with respect to the host. In synchronous mode, the host and GPU device are synchronized prior to the operation returning.

Arguments `LS` (`SUNLinearSolver`) a `SUNLinSol_MagmaDense` object
`onoff` (`booleantype`) set to 0 for synchronous mode, or 1 for asynchronous mode

Return value None

Notes The default is asynchronous mode.

9.13.3 SUNLinearSolver_MagmaDense content

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

```
struct _SUNLinearSolverContent_MagmaDense {
    int          last_flag;
    booleantype  async;
    sunindextype N;
    SUNMemory    pivots;
    SUNMemory    pivotsarr;
    SUNMemory    dpivotsarr;
    SUNMemory    infoarr;
    SUNMemory    rhsarr;
    SUNMemoryHelper memhelp;
    magma_queue_t q;
};
```

9.14 The SUNLinearSolver_OneMklDense Implementation

The `SUNLinearSolver_OneMklDense` implementation of the `SUNLINSOL` class interfaces to the direct linear solvers from the [Intel oneAPI Math Kernel Library \(oneMKL\)](#) for solving dense systems of block-diagonal systems with dense blocks. This linear solver is best paired with the `SUNMatrix_OneMklDense` matrix.

The header file to include when using this class is `sunlinsol/sunlinsol_onemkldense.h`. The installed library to link to is `libsundials_sunlinsolonemkldense.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The `SUNLinearSolver_OneMklDense` class is experimental and subject to change.



9.14.1 SUNLinearSolver_OneMklDense Functions

The `SUNLinearSolver_OneMklDense` class defines implementations of all “direct” linear solver operations listed in Sections [9.1.1-9.1.3](#):

- `SUNLinSolGetType_OneMklDense` – returns `SUNLINEARSOLVER_ONEMKLDENSE`
- `SUNLinSolInitialize_OneMklDense`
- `SUNLinSolSetup_OneMklDense`
- `SUNLinSolSolve_OneMklDense`
- `SUNLinSolLastFlag_OneMklDense`
- `SUNLinSolFree_OneMklDense`

In addition, the class provides the following user-callable routines:

<code>SUNLinSol_OneMklDense</code>	
Call	<code>LS = SUNLinSol_OneMklDense(y, A);</code>
Description	The function <code>SUNLinSol_OneMklDense</code> creates and allocates memory for a <code>SUNLINSOL</code> object.
Arguments	<code>y</code> (<code>N.Vector</code>) a vector for checking compatibility with the solver <code>A</code> (<code>SUNMatrix</code>) a <code>SUNMATRIX_ONEMKLDENSE</code> matrix for checking compatibility with the solver
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>A</code> or <code>y</code> are incompatible then this routine will return <code>NULL</code> .
Notes	This routine analyzes the input matrix and vector to determine the linear system size and to assess compatibility with the solver.

9.14.2 SUNLinearSolver_OneMklDense Usage Notes

The `SUNLinearSolver_OneMklDense` class only supports 64-bit indexing, thus `SUNDIALS` must be built for 64-bit indexing to use this class.

When using the `SUNLinearSolver_OneMklDense` class with a `SUNDIALS` package (e.g. `CVODE`), the queue given to matrix is also used for the linear solver.

9.15 The SUNLinearSolver_SPGMR implementation

This section describes the SUNLINSOL implementation of the SPGMR (Scaled, Preconditioned, Generalized Minimum Residual [38]) iterative linear solver. The SUNLINSOL_SPGMR module is designed to be compatible with any NVECTOR implementation 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.

To access the SUNLINSOL_SPGMR module, include the header file `sunlinsol/sunlinsol_spgmr.h`. We note that the SUNLINSOL_SPGMR module is accessible from SUNDIALS packages *without* separately linking to the `libsundials_sunlinsolspgmr` module library.

9.15.1 SUNLinearSolver_SPGMR description

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.

9.15.2 SUNLinearSolver_SPGMR functions

The SUNLINSOL_SPGMR module provides the following user-callable constructor for creating a SUNLinearSolver object.

<code>SUNLinSol_SPGMR</code>	
Call	<code>LS = SUNLinSol_SPGMR(y, pretype, max1);</code>
Description	The function <code>SUNLinSol_SPGMR</code> creates and allocates memory for a SPGMR <code>SUNLinearSolver</code> object.
Arguments	<p><code>y</code> (N_Vector) a template for cloning vectors needed within the solver</p> <p><code>pretype</code> (int) flag indicating the desired type of preconditioning, allowed values are:</p> <ul style="list-style-type: none"> • <code>PREC_NONE</code> (0) • <code>PREC_LEFT</code> (1) • <code>PREC_RIGHT</code> (2) • <code>PREC_BOTH</code> (3) <p>Any other integer input will result in the default (no preconditioning).</p> <p><code>max1</code> (int) the number of Krylov basis vectors to use. Values ≤ 0 will result in the default value (5).</p>
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>y</code> is incompatible then this routine will return <code>NULL</code> .

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.

Deprecated Name For backward compatibility, the wrapper function `SUNSPGMR` with identical input and output arguments is also provided.

F2003 Name `FSUNLinSol_SPGMR`

The `SUNLINSOL_SPGMR` module defines implementations of all “iterative” linear solver operations listed in Sections 9.1.1 – 9.1.3:

- `SUNLinSolGetType_SPGMR`
- `SUNLinSolInitialize_SPGMR`
- `SUNLinSolSetATimes_SPGMR`
- `SUNLinSolSetPreconditioner_SPGMR`
- `SUNLinSolSetScalingVectors_SPGMR`
- `SUNLinSolSetZeroGuess_SPGMR` – note the solver assumes a non-zero guess by default and the zero guess flag is reset to `SUNFALSE` after each call to `SUNLinSolSolve_SPGMR`.
- `SUNLinSolSetup_SPGMR`
- `SUNLinSolSolve_SPGMR`
- `SUNLinSolNumIters_SPGMR`
- `SUNLinSolResNorm_SPGMR`
- `SUNLinSolResid_SPGMR`
- `SUNLinSolLastFlag_SPGMR`
- `SUNLinSolSpace_SPGMR`
- `SUNLinSolFree_SPGMR`

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’ to the function name.

The `SUNLINSOL_SPGMR` module also defines the following additional user-callable functions.

`SUNLinSol_SPGMRSetPrecType`

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

Deprecated Name For backward compatibility, the wrapper function `SUNSPGMRSetPrecType` with identical input and output arguments is also provided.

F2003 Name `FSUNLinSol_SPGMRSetPrecType`

`SUNLinSol_SPGMRSetGSType`

Call `retval = SUNLinSol_SPGMRSetGSType(LS, gstype);`

Description The function `SUNLinSol_SPGMRSetPrecType` 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:

- `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`.

Deprecated Name For backward compatibility, the wrapper function `SUNSPGMRSetGSType` with identical input and output arguments is also provided.

F2003 Name `FSUNLinSol_SPGMRSetGSType`

`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`.

Deprecated Name For backward compatibility, the wrapper function `SUNSPGMRSetMaxRestarts` with identical input and output arguments is also provided.

F2003 Name `FSUNLinSol_SPGMRSetMaxRestarts`

`SUNLinSolSetInfoFile_SPGMR`

Call `retval = SUNLinSolSetInfoFile_SPGMR(LS, info_file);`

Description The function `SUNLinSolSetInfoFile_SPGMR` sets the output file where all informative (non-error) messages should be directed.

Arguments `LS` (`SUNLinearSolver`) a `SUNNONLINSOL` object
`info_file` (`FILE*`) pointer to output file (`stdout` by default); a `NULL` input will disable output

Return value The return value is

- `SUNLS_SUCCESS` if successful
- `SUNLS_MEM_NULL` if the `SUNLinearSolver` memory was `NULL`
- `SUNLS_ILL_INPUT` if `SUNDIALS` was not built with monitoring enabled

Notes This function is intended for users that wish to monitor the linear solver progress. By default, the file pointer is set to `stdout`.

SUNDIALS must be built with the CMake option `SUNDIALS_BUILD_WITH_MONITORING`, to utilize this function. See section [A.1.2](#) for more information.

F2003 Name FSUNLinSolSetInfoFile_SPGMR

SUNLinSolSetPrintLevel_SPGMR

Call `retval = SUNLinSolSetPrintLevel_SPGMR(NLS, print_level);`

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

Arguments LS (SUNLinearSolver) a SUNNONLINSOL object
`print_level` (int) flag indicating level of verbosity; must be one of:

- 0, no information is printed (default)
- 1, for each linear iteration the residual norm is printed

Return value The return value is

- `SUNLS_SUCCESS` if successful
- `SUNLS_MEM_NULL` if the SUNLinearSolver memory was NULL
- `SUNLS_ILL_INPUT` if SUNDIALS was not built with monitoring enabled, or the print level value was invalid

Notes This function is intended for users that wish to monitor the linear solver progress. By default, the print level is 0.

SUNDIALS must be built with the CMake option `SUNDIALS_BUILD_WITH_MONITORING`, to utilize this function. See section [A.1.2](#) for more information.

F2003 Name FSUNLinSolSetPrintLevel_SPGMR

9.15.3 SUNLinearSolver_SPGMR Fortran interfaces

The `SUNLINSOL_SPGMR` module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The `fsunlinsol_spgmr_mod` FORTRAN module defines interfaces to all `SUNLINSOL_SPGMR` C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function `SUNLinSol_SPGMR` is interfaced as `FSUNLinSol_SPGMR`.

The FORTRAN 2003 `SUNLINSOL_SPGMR` interface module can be accessed with the `use` statement, i.e. `use fsunlinsol_spgmr_mod`, and linking to the library `libsundials_fsunlinsol_spgmr_mod.lib` in addition to the C library. For details on where the library and module file `fsunlinsol_spgmr_mod.mod` are installed see [Appendix A](#). We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators *without* separately linking to the `libsundials_fsunlinsol_spgmr_mod` library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 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 SUNLINSOL_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_SPGMRSetGStype 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.

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 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.

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 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_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 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_SPGMRSetMaxRestarts` for complete further documentation of this routine.

9.15.4 SUNLinearSolver_SPGMR content

The `SUNLINSOL_SPGMR` module defines the *content* field of a `SUNLinearSolver` as the following structure:

```
struct _SUNLinearSolverContent_SPGMR {
    int maxl;
    int pretype;
    int gstype;
    int max_restarts;
    booleantype zeroguess;
    int numiters;
    realtype resnorm;
    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;
    int print_level;
    FILE* info_file;
};
```

These entries of the *content* field contain the following information:

<code>maxl</code>	- number of GMRES basis vectors to use (default is 5),
<code>pretype</code>	- flag for type of preconditioning to employ (default is none),
<code>gstype</code>	- flag for type of Gram-Schmidt orthogonalization (default is modified Gram-Schmidt),
<code>max_restarts</code>	- number of GMRES restarts to allow (default is 0),
<code>numiters</code>	- number of iterations from the most-recent solve,
<code>resnorm</code>	- final linear residual norm from the most-recent solve,
<code>last_flag</code>	- last error return flag from an internal function,
<code>ATimes</code>	- function pointer to perform Av product,
<code>ATData</code>	- pointer to structure for <code>ATimes</code> ,
<code>Psetup</code>	- function pointer to preconditioner setup routine,
<code>Psolve</code>	- function pointer to preconditioner solve routine,
<code>PData</code>	- pointer to structure for <code>Psetup</code> and <code>Psolve</code> ,
<code>s1, s2</code>	- vector pointers for supplied scaling matrices (default is NULL),

- 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.

9.16.2 SUNLinearSolver_SPFGMR functions

The SUNLINSOL_SPFGMR module provides the following user-callable constructor for creating a SUNLinearSolver object.

SUNLinSol_SPFGMR

Call `LS = SUNLinSol_SPFGMR(y, pretype, max1);`

Description The function SUNLinSol_SPFGMR creates and allocates memory for a SPFGMR SUNLinearSolver object.

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).

`max1` (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_SPFGMR object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.

F2003 Name FSUNLinSol_SPFGMR

SUNSPFGMR The SUNLINSOL_SPFGMR module defines implementations of all “iterative” linear solver operations listed in Sections 9.1.1 – 9.1.3:

- SUNLinSolGetType_SPFGMR
- SUNLinSolInitialize_SPFGMR
- SUNLinSolSetATimes_SPFGMR
- SUNLinSolSetPreconditioner_SPFGMR
- SUNLinSolSetScalingVectors_SPFGMR
- SUNLinSolSetZeroGuess_SPFGMR – note the solver assumes a non-zero guess by default and the zero guess flag is reset to SUNFALSE after each call to SUNLinSolSolve_SPFGMR.
- SUNLinSolSetup_SPFGMR

- SUNLinSolSolve_SPFGMR
- SUNLinSolNumIters_SPFGMR
- SUNLinSolResNorm_SPFGMR
- SUNLinSolResid_SPFGMR
- SUNLinSolLastFlag_SPFGMR
- SUNLinSolSpace_SPFGMR
- SUNLinSolFree_SPFGMR

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’ to the function name.

The SUNLINSOL_SPFGMR module also defines the following additional user-callable functions.

SUNLinSol_SPFGMRSetPrecType

Call	<code>retval = SUNLinSol_SPFGMRSetPrecType(LS, pretype);</code>
Description	The function <code>SUNLinSol_SPFGMRSetPrecType</code> updates the type of preconditioning to use in the <code>SUNLINSOL_SPFGMR</code> object.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) the <code>SUNLINSOL_SPFGMR</code> object to update <code>pretype</code> (<code>int</code>) flag indicating the desired type of preconditioning, allowed values match those discussed in <code>SUNLinSol_SPFGMR</code> .
Return value	This routine will return with one of the error codes <code>SUNLS_ILL_INPUT</code> (illegal <code>pretype</code>), <code>SUNLS_MEM_NULL</code> (<code>S</code> is <code>NULL</code>) or <code>SUNLS_SUCCESS</code> .
Deprecated Name	For backward compatibility, the wrapper function <code>SUNSPFGMRSetPrecType</code> with identical input and output arguments is also provided.
F2003 Name	<code>FSUNLinSol_SPFGMRSetPrecType</code>

SUNLinSol_SPFGMRSetGSType

Call	<code>retval = SUNLinSol_SPFGMRSetGSType(LS, gstype);</code>
Description	The function <code>SUNLinSol_SPFGMRSetGSType</code> sets the type of Gram-Schmidt orthogonalization to use in the <code>SUNLINSOL_SPFGMR</code> object.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) the <code>SUNLINSOL_SPFGMR</code> object to update <code>gstype</code> (<code>int</code>) flag indicating the desired orthogonalization algorithm; allowed values are: <ul style="list-style-type: none"> • <code>MODIFIED_GS</code> (1) • <code>CLASSICAL_GS</code> (2) Any other integer input will result in a failure, returning error code <code>SUNLS_ILL_INPUT</code> .
Return value	This routine will return with one of the error codes <code>SUNLS_ILL_INPUT</code> (illegal <code>pretype</code>), <code>SUNLS_MEM_NULL</code> (<code>S</code> is <code>NULL</code>) or <code>SUNLS_SUCCESS</code> .
Deprecated Name	For backward compatibility, the wrapper function <code>SUNSPFGMRSetGSType</code> with identical input and output arguments is also provided.
F2003 Name	<code>FSUNLinSol_SPFGMRSetGSType</code>

SUNLinSol_SPGMRSetMaxRestarts

Call	<code>retval = SUNLinSol_SPGMRSetMaxRestarts(LS, maxrs);</code>
Description	The function <code>SUNLinSol_SPGMRSetMaxRestarts</code> sets the number of GMRES restarts to allow in the <code>SUNLINSOL_SPGMR</code> object.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) the <code>SUNLINSOL_SPGMR</code> object to update <code>maxrs</code> (<code>int</code>) 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 <code>SUNLS_MEM_NULL</code> (S is NULL) or <code>SUNLS_SUCCESS</code> .
Deprecated Name	For backward compatibility, the wrapper function <code>SUNSPFGMRSetMaxRestarts</code> with identical input and output arguments is also provided.
F2003 Name	<code>FSUNLinSol_SPGMRSetMaxRestarts</code>

SUNLinSolSetInfoFile_SPGMR

Call	<code>retval = SUNLinSolSetInfoFile_SPGMR(LS, info_file);</code>
Description	The function <code>SUNLinSolSetInfoFile_SPGMR</code> sets the output file where all informative (non-error) messages should be directed.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) a <code>SUNNONLINSOL</code> object <code>info_file</code> (<code>FILE*</code>) pointer to output file (<code>stdout</code> by default); a NULL input will disable output
Return value	The return value is <ul style="list-style-type: none"> • <code>SUNLS_SUCCESS</code> if successful • <code>SUNLS_MEM_NULL</code> if the <code>SUNLinearSolver</code> memory was NULL • <code>SUNLS_ILL_INPUT</code> if <code>SUNDIALS</code> was not built with monitoring enabled
Notes	This function is intended for users that wish to monitor the linear solver progress. By default, the file pointer is set to <code>stdout</code> . SUNDIALS must be built with the CMake option <code>SUNDIALS_BUILD_WITH_MONITORING</code>, to utilize this function. See section A.1.2 for more information.
F2003 Name	<code>FSUNLinSolSetInfoFile_SPGMR</code>

SUNLinSolSetPrintLevel_SPGMR

Call	<code>retval = SUNLinSolSetPrintLevel_SPGMR(NLS, print_level);</code>
Description	The function <code>SUNLinSolSetPrintLevel_SPGMR</code> specifies the level of verbosity of the output.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) a <code>SUNNONLINSOL</code> object <code>print_level</code> (<code>int</code>) flag indicating level of verbosity; must be one of: <ul style="list-style-type: none"> • 0, no information is printed (default) • 1, for each linear iteration the residual norm is printed
Return value	The return value is <ul style="list-style-type: none"> • <code>SUNLS_SUCCESS</code> if successful • <code>SUNLS_MEM_NULL</code> if the <code>SUNLinearSolver</code> memory was NULL • <code>SUNLS_ILL_INPUT</code> if <code>SUNDIALS</code> was not built with monitoring enabled, or the print level value was invalid

Notes This function is intended for users that wish to monitor the linear solver progress. By default, the print level is 0.

SUNDIALS **must be built with the CMake option** `SUNDIALS_BUILD_WITH_MONITORING`, **to utilize this function.** See section [A.1.2](#) for more information.

F2003 Name `FSUNLinSolSetPrintLevel_SPGMR`

9.16.3 SUNLinearSolver_SPGMR Fortran interfaces

The `SUNLINSOL_SPGMR` module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The `fsunlinsol_spgmr_mod` FORTRAN module defines interfaces to all `SUNLINSOL_SPGMR` C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function `SUNLinSol_SPGMR` is interfaced as `FSUNLinSol_SPGMR`.

The FORTRAN 2003 `SUNLINSOL_SPGMR` interface module can be accessed with the `use` statement, i.e. `use fsunlinsol_spgmr_mod`, and linking to the library `libsundials_fsunlinsolspgmr_mod.lib` in addition to the C library. For details on where the library and module file `fsunlinsol_spgmr_mod.mod` are installed see [Appendix A](#). We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators *without* separately linking to the `libsundials_fsunlinsolspgmr_mod` library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 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 `SUNLINSOL_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.

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 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.

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 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.

FSUNSPFGMRSETPRECTYPE

Call `FSUNSPFGMRSETPRECTYPE(code, pretype, ier)`

Description The function `FSUNSPFGMRSETPRECTYPE` 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.

FSUNMASSSPFGMRSETPRECTYPE

Call `FSUNMASSSPFGMRSETPRECTYPE(prectype, 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 `KINSOL`, 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.

9.16.4 SUNLinearSolver_SPFGMR content

The `SUNLINSOL_SPFGMR` module defines the *content* field of a `SUNLinearSolver` as the following structure:

```
struct _SUNLinearSolverContent_SPFGMR {
    int maxl;
    int pretype;
    int gstype;
    int max_restarts;
    booleantype zeroguess;
    int numiters;
    realtype resnorm;
    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;
int      print_level;
FILE*    info_file;
};

```

These entries of the *content* field contain the following information:

max1 - number of FGMRES 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 FGMRES restarts to allow (default is 0),
numiters - number of iterations from the most-recent solve,
resnorm - final linear residual norm from the most-recent solve,
last_flag - last error return flag from an internal function,
ATimes - function pointer to perform Av product,
ATData - pointer to structure for **ATimes**,
Psetup - function pointer to preconditioner setup routine,
Psolve - function pointer to preconditioner solve routine,
PData - pointer to structure for **Psetup** and **Psolve**,
s1, s2 - vector pointers for supplied scaling matrices (default is NULL),
V - the array of Krylov basis vectors $v_1, \dots, v_{\max1+1}$, stored in $V[0], \dots, V[\max1]$. Each v_i is a vector of type **NVECTOR**,
Z - the array of preconditioned Krylov basis vectors $z_1, \dots, z_{\max1+1}$, stored in $Z[0], \dots, Z[\max1]$. Each z_i is a vector of type **NVECTOR**,
Hes - the $(\max1 + 1) \times \max1$ Hessenberg matrix. It is stored row-wise so that the (i,j) th element is given by $Hes[i][j]$,
givens - a length $2*\max1$ array which represents the Givens rotation matrices that arise in the FGMRES algorithm. These matrices are F_0, F_1, \dots, 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 `realt` values used to hold “short” vectors (e.g. y and g),
vtemp - temporary vector storage.
print_level - controls the amount of information to be printed to the info file
info_file - the file where all informative (non-error) messages will be directed

9.17 The SUNLinearSolver_SPBCGS implementation

This section describes the SUNLINSOL implementation of the SPBCGS (Scaled, Preconditioned, Bi-Conjugate Gradient, Stabilized [40]) iterative linear solver. The SUNLINSOL_SPBCGS module is designed to be compatible with any NVECTOR implementation 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.

To access the SUNLINSOL_SPBCGS module, include the header file `sunlinsol/sunlinsol_spbcgs.h`. We note that the SUNLINSOL_SPBCGS module is accessible from SUNDIALS packages *without* separately linking to the `libsundials_sunlinsolspbcgs` module library.

9.17.1 SUNLinearSolver_SPBCGS description

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.

9.17.2 SUNLinearSolver_SPBCGS functions

The SUNLINSOL_SPBCGS module provides the following user-callable constructor for creating a `SUNLinearSolver` object.

<code>SUNLinSol_SPBCGS</code>	
Call	<code>LS = SUNLinSol_SPBCGS(y, pretype, maxl);</code>
Description	The function <code>SUNLinSol_SPBCGS</code> creates and allocates memory for a SPBCGS <code>SUNLinearSolver</code> object.
Arguments	<p><code>y</code> (<code>N_Vector</code>) a template for cloning vectors needed within the solver</p> <p><code>pretype</code> (<code>int</code>) flag indicating the desired type of preconditioning, allowed values are:</p> <ul style="list-style-type: none"> • <code>PREC_NONE</code> (0)

	<ul style="list-style-type: none"> • <code>PREC_LEFT</code> (1) • <code>PREC_RIGHT</code> (2) • <code>PREC_BOTH</code> (3)
	Any other integer input will result in the default (no preconditioning).
	<code>max1</code> (<code>int</code>) the number of linear iterations to allow. Values ≤ 0 will result in the default value (5).
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>y</code> is incompatible then this routine will return <code>NULL</code> .
Notes	<p>This routine will perform consistency checks to ensure that it is called with a consistent <code>NVECTOR</code> implementation (i.e. that it supplies the requisite vector operations). If <code>y</code> is incompatible, then this routine will return <code>NULL</code>.</p> <p>We note that some <code>SUNDIALS</code> solvers are designed to only work with left preconditioning (<code>IDA</code> and <code>IDAS</code>) and others with only right preconditioning (<code>KINSOL</code>). While it is possible to configure a <code>SUNLINSOL_SPBCGS</code> object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.</p> <p>With <code>PREC_RIGHT</code> or <code>PREC_BOTH</code> the initial guess must be zero (use <code>SUNLinSolSetZeroGuess</code> to indicate the initial guess is zero).</p>
Deprecated Name	For backward compatibility, the wrapper function <code>SUNSPBCGS</code> with identical input and output arguments is also provided.
F2003 Name	<code>FSUNLinSol_SPBCGS</code>

The `SUNLINSOL_SPBCGS` module defines implementations of all “iterative” linear solver operations listed in Sections 9.1.1 – 9.1.3:

- `SUNLinSolGetType_SPBCGS`
- `SUNLinSolInitialize_SPBCGS`
- `SUNLinSolSetATimes_SPBCGS`
- `SUNLinSolSetPreconditioner_SPBCGS`
- `SUNLinSolSetScalingVectors_SPBCGS`
- `SUNLinSolSetZeroGuess_SPBCGS` – note the solver assumes a non-zero guess by default and the zero guess flag is reset to `SUNFALSE` after each call to `SUNLinSolSolve_SPBCGS`.
- `SUNLinSolSetup_SPBCGS`
- `SUNLinSolSolve_SPBCGS`
- `SUNLinSolNumIters_SPBCGS`
- `SUNLinSolResNorm_SPBCGS`
- `SUNLinSolResid_SPBCGS`
- `SUNLinSolLastFlag_SPBCGS`
- `SUNLinSolSpace_SPBCGS`
- `SUNLinSolFree_SPBCGS`

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’ to the function name.

The `SUNLINSOL_SPBCGS` module also defines the following additional user-callable functions.

SUNLinSol_SPBCGSSetPrecType

Call	<code>retval = SUNLinSol_SPBCGSSetPrecType(LS, pretype);</code>
Description	The function <code>SUNLinSol_SPBCGSSetPrecType</code> updates the type of preconditioning to use in the <code>SUNLINSOL_SPBCGS</code> object.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) the <code>SUNLINSOL_SPBCGS</code> object to update <code>pretype</code> (<code>int</code>) flag indicating the desired type of preconditioning, allowed values match those discussed in <code>SUNLinSol_SPBCGS</code> .
Return value	This routine will return with one of the error codes <code>SUNLS_ILL_INPUT</code> (illegal <code>pretype</code>), <code>SUNLS_MEM_NULL</code> (<code>S</code> is <code>NULL</code>) or <code>SUNLS_SUCCESS</code> .
Deprecated Name	For backward compatibility, the wrapper function <code>SUNSPBCGSSetPrecType</code> with identical input and output arguments is also provided.
F2003 Name	<code>FSUNLinSol_SPBCGSSetPrecType</code>

SUNLinSol_SPBCGSsetMax1

Call	<code>retval = SUNLinSol_SPBCGSsetMax1(LS, max1);</code>
Description	The function <code>SUNLinSol_SPBCGSsetMax1</code> updates the number of linear solver iterations to allow.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) the <code>SUNLINSOL_SPBCGS</code> object to update <code>max1</code> (<code>int</code>) flag indicating the number of iterations to allow. Values ≤ 0 will result in the default value (5).
Return value	This routine will return with one of the error codes <code>SUNLS_MEM_NULL</code> (<code>S</code> is <code>NULL</code>) or <code>SUNLS_SUCCESS</code> .
Deprecated Name	For backward compatibility, the wrapper function <code>SUNSPBCGSsetMax1</code> with identical input and output arguments is also provided.
F2003 Name	<code>FSUNLinSol_SPBCGSsetMax1</code>

SUNLinSolSetInfoFile_SPBCGS

Call	<code>retval = SUNLinSolSetInfoFile_SPBCGS(LS, info_file);</code>
Description	The function <code>SUNLinSolSetInfoFile_SPBCGS</code> sets the output file where all informative (non-error) messages should be directed.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) a <code>SUNNONLINSOL</code> object <code>info_file</code> (<code>FILE*</code>) pointer to output file (<code>stdout</code> by default); a <code>NULL</code> input will disable output
Return value	The return value is <ul style="list-style-type: none"> • <code>SUNLS_SUCCESS</code> if successful • <code>SUNLS_MEM_NULL</code> if the <code>SUNLinearSolver</code> memory was <code>NULL</code> • <code>SUNLS_ILL_INPUT</code> if <code>SUNDIALS</code> was not built with monitoring enabled
Notes	This function is intended for users that wish to monitor the linear solver progress. By default, the file pointer is set to <code>stdout</code> . SUNDIALS must be built with the CMake option <code>SUNDIALS_BUILD_WITH_MONITORING</code>, to utilize this function. See section A.1.2 for more information.
F2003 Name	<code>FSUNLinSolSetInfoFile_SPBCGS</code>

SUNLinSolSetPrintLevel_SPBCGS

Call	<code>retval = SUNLinSolSetPrintLevel_SPBCGS(NLS, print_level);</code>
Description	The function <code>SUNLinSolSetPrintLevel_SPBCGS</code> specifies the level of verbosity of the output.
Arguments	<p><code>LS</code> (<code>SUNLinearSolver</code>) a <code>SUNNONLINSOL</code> object</p> <p><code>print_level</code> (<code>int</code>) flag indicating level of verbosity; must be one of:</p> <ul style="list-style-type: none"> • 0, no information is printed (default) • 1, for each linear iteration the residual norm is printed
Return value	<p>The return value is</p> <ul style="list-style-type: none"> • <code>SUNLS_SUCCESS</code> if successful • <code>SUNLS_MEM_NULL</code> if the <code>SUNLinearSolver</code> memory was <code>NULL</code> • <code>SUNLS_ILL_INPUT</code> if <code>SUNDIALS</code> was not built with monitoring enabled, or the print level value was invalid
Notes	<p>This function is intended for users that wish to monitor the linear solver progress. By default, the print level is 0.</p> <p>SUNDIALS must be built with the CMake option <code>SUNDIALS_BUILD_WITH_MONITORING</code>, to utilize this function. See section A.1.2 for more information.</p>
F2003 Name	<code>FSUNLinSolSetPrintLevel_SPBCGS</code>

9.17.3 SUNLinearSolver_SPBCGS Fortran interfaces

The `SUNLINSOL_SPBCGS` module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTTRAN 2003 interface module

The `fsunlinsol_spbcgs_mod` FORTRAN module defines interfaces to all `SUNLINSOL_SPBCGS` C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading ‘F’. For example, the function `SUNLinSol_SPBCGS` is interfaced as `FSUNLinSol_SPBCGS`.

The FORTRAN 2003 `SUNLINSOL_SPBCGS` interface module can be accessed with the `use` statement, i.e. `use fsunlinsol_spbcgs_mod`, and linking to the library `libsundials_fsunlinsolspbcgs_mod.lib` in addition to the C library. For details on where the library and module file `fsunlinsol_spbcgs_mod.mod` are installed see [Appendix A](#). We note that the module is accessible from the FORTRAN 2003 `SUNDIALS` integrators *without* separately linking to the `libsundials_fsunlinsolspbcgs_mod` library.

FORTTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the `SUNLINSOL_SPBCGS` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

FSUNSPBCGSINIT

Call	<code>FSUNSPBCGSINIT(code, pretype, maxl, ier)</code>
Description	The function <code>FSUNSPBCGSINIT</code> can be called for Fortran programs to create a <code>SUNLINSOL_SPBCGS</code> object.
Arguments	<p><code>code</code> (<code>int*</code>) is an integer input specifying the solver id (1 for <code>CVODE</code>, 2 for <code>IDA</code>, 3 for <code>KINSOL</code>, and 4 for <code>ARKODE</code>).</p>

`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 `SUNLINSOL_SPBCGS` module includes a Fortran-callable function for creating a `SUNLinearSolver` mass matrix solver object.

FSUNMASSSPBCGSINIT

Call `FSUNMASSSPBCGSINIT(pretype, maxl, ier)`

Description The function `FSUNMASSSPBCGSINIT` can be called for Fortran programs to create a `SUNLINSOL_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 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_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 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_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 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_SPBCGSsetMaxl` for complete further documentation of this routine.

FSUNMASSPBCGSSETMAXL

Call `FSUNMASSPBCGSSETMAXL(maxl, ier)`

Description The function `FSUNMASSPBCGSSETMAXL` 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 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_SPBCGSsetMaxl` for complete further documentation of this routine.

9.17.4 SUNLinearSolver_SPBCGS content

The `SUNLINSOL_SPBCGS` module defines the *content* field of a `SUNLinearSolver` as the following structure:

```
struct _SUNLinearSolverContent_SPBCGS {
    int maxl;
    int pretype;
    booleantype zeroguess;
    int numiters;
    realtype resnorm;
    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;
    int print_level;
    FILE* info_file;
};
```

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.
print_level - controls the amount of information to be printed to the info file
info_file - the file where all informative (non-error) messages will be directed

9.18 The SUNLinearSolver_SPTFQMR implementation

This section describes the SUNLINSOL implementation of the SPTFQMR (Scaled, Preconditioned, Transpose-Free Quasi-Minimum Residual [23]) iterative linear solver. The SUNLINSOL_SPTFQMR module is designed to be compatible with any NVECTOR implementation that supports a minimal subset of operations (**N_VClone**, **N_VDotProd**, **N_VScale**, **N_VLinearSum**, **N_VProd**, **N_VConst**, **N_VDiv**, and **N_VDestroy**). Unlike the SPGMR and SPFGMR algorithms, SPTFQMR requires a fixed amount of memory that does not increase with the number of allowed iterations.

To access the SUNLINSOL_SPTFQMR module, include the header file `sunlinsol/sunlinsol_sptfqmr.h`. We note that the SUNLINSOL_SPTFQMR module is accessible from SUNDIALS packages *without* separately linking to the `libsundials_sunlinsolsptfqmr` module library.

9.18.1 SUNLinearSolver_SPTFQMR description

This solver is constructed to perform the following operations:

- During construction all NVECTOR solver data is allocated, with vectors cloned from a template NVECTOR that is input, and default solver parameters are set.
- User-facing “set” routines may be called to modify default solver parameters.
- Additional “set” routines are called by the SUNDIALS solver that interfaces with SUNLINSOL_SPTFQMR to supply the **ATimes**, **PSetup**, and **Psolve** function pointers and **s1** and **s2** scaling vectors.
- In the “initialize” call, the solver parameters are checked for validity.
- In the “setup” call, any non-NULL **PSetup** function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic **PSetup** function and the solver-specific routine (solver-supplied or user-supplied).
- In the “solve” call the TFQMR iteration is performed. This will include scaling and preconditioning if those options have been supplied.

9.18.2 SUNLinearSolver_SPTFQMR functions

The SUNLINSOL_SPTFQMR module provides the following user-callable constructor for creating a SUNLinearSolver object.

SUNLinSol_SPTFQMR	
Call	<code>LS = SUNLinSol_SPTFQMR(y, pretype, maxl);</code>
Description	The function <code>SUNLinSol_SPTFQMR</code> creates and allocates memory for a SPTFQMR <code>SUNLinearSolver</code> object.
Arguments	<p><code>y</code> (<code>N_Vector</code>) a template for cloning vectors needed within the solver</p> <p><code>pretype</code> (<code>int</code>) flag indicating the desired type of preconditioning, allowed values are:</p> <ul style="list-style-type: none"> • <code>PREC_NONE</code> (0) • <code>PREC_LEFT</code> (1) • <code>PREC_RIGHT</code> (2) • <code>PREC_BOTH</code> (3) <p>Any other integer input will result in the default (no preconditioning).</p> <p><code>maxl</code> (<code>int</code>) the number of linear iterations to allow. Values ≤ 0 will result in the default value (5).</p>
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>y</code> is incompatible then this routine will return <code>NULL</code> .
Notes	<p>This routine will perform consistency checks to ensure that it is called with a consistent <code>NVECTOR</code> implementation (i.e. that it supplies the requisite vector operations). If <code>y</code> is incompatible, then this routine will return <code>NULL</code>.</p> <p>We note that some <code>SUNDIALS</code> solvers are designed to only work with left preconditioning (<code>IDA</code> and <code>IDAS</code>) and others with only right preconditioning (<code>KINSOL</code>). While it is possible to configure a <code>SUNLINSOL_SPTFQMR</code> object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.</p> <p>With <code>PREC_RIGHT</code> or <code>PREC_BOTH</code> the initial guess must be zero (use <code>SUNLinSolSetZeroGuess</code> to indicate the initial guess is zero).</p>
Deprecated Name	For backward compatibility, the wrapper function <code>SUNSPTFQMR</code> with identical input and output arguments is also provided.
F2003 Name	<code>FSUNLinSol_SPTFQMR</code>

The `SUNLINSOL_SPTFQMR` module defines implementations of all “iterative” linear solver operations listed in Sections [9.1.1](#) – [9.1.3](#):

- `SUNLinSolGetType_SPTFQMR`
- `SUNLinSolInitialize_SPTFQMR`
- `SUNLinSolSetATimes_SPTFQMR`
- `SUNLinSolSetPreconditioner_SPTFQMR`
- `SUNLinSolSetScalingVectors_SPTFQMR`
- `SUNLinSolSetZeroGuess_SPTFQMR` – note the solver assumes a non-zero guess by default and the zero guess flag is reset to `SUNFALSE` after each call to `SUNLinSolSolve_SPTFQMR`.
- `SUNLinSolSetup_SPTFQMR`
- `SUNLinSolSolve_SPTFQMR`

- SUNLinSolNumIters_SPTFQMR
- SUNLinSolResNorm_SPTFQMR
- SUNLinSolResid_SPTFQMR
- SUNLinSolLastFlag_SPTFQMR
- SUNLinSolSpace_SPTFQMR
- SUNLinSolFree_SPTFQMR

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’ to the function name.

The SUNLINSOL_SPTFQMR module also defines the following additional user-callable functions.

SUNLinSol_SPTFQMRSetPrecType

Call	<code>retval = SUNLinSol_SPTFQMRSetPrecType(LS, pretype);</code>
Description	The function <code>SUNLinSol_SPTFQMRSetPrecType</code> updates the type of preconditioning to use in the <code>SUNLINSOL_SPTFQMR</code> object.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) the <code>SUNLINSOL_SPTFQMR</code> object to update <code>pretype</code> (<code>int</code>) flag indicating the desired type of preconditioning, allowed values match those discussed in <code>SUNLinSol_SPTFQMR</code> .
Return value	This routine will return with one of the error codes <code>SUNLS_ILL_INPUT</code> (illegal <code>pretype</code>), <code>SUNLS_MEM_NULL</code> (<code>S</code> is <code>NULL</code>) or <code>SUNLS_SUCCESS</code> .
Deprecated Name	For backward compatibility, the wrapper function <code>SUNSPTFQMRSetPrecType</code> with identical input and output arguments is also provided.
F2003 Name	<code>FSUNLinSol_SPTFQMRSetPrecType</code>

SUNLinSol_SPTFQMRSetMax1

Call	<code>retval = SUNLinSol_SPTFQMRSetMax1(LS, max1);</code>
Description	The function <code>SUNLinSol_SPTFQMRSetMax1</code> updates the number of linear solver iterations to allow.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) the <code>SUNLINSOL_SPTFQMR</code> object to update <code>max1</code> (<code>int</code>) flag indicating the number of iterations to allow; values ≤ 0 will result in the default value (5)
Return value	This routine will return with one of the error codes <code>SUNLS_MEM_NULL</code> (<code>S</code> is <code>NULL</code>) or <code>SUNLS_SUCCESS</code> .
F2003 Name	<code>FSUNLinSol_SPTFQMRSetMax1</code>
	<code>SUNSPTFQMRSetMax1</code>

SUNLinSolSetInfoFile_SPTFQMR

Call	<code>retval = SUNLinSolSetInfoFile_SPTFQMR(LS, info_file);</code>
Description	The function <code>SUNLinSolSetInfoFile_SPTFQMR</code> sets the output file where all informative (non-error) messages should be directed.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) a <code>SUNNONLINSOL</code> object <code>info_file</code> (<code>FILE*</code>) pointer to output file (<code>stdout</code> by default); a <code>NULL</code> input will disable output
Return value	The return value is

- SUNLS_SUCCESS if successful
- SUNLS_MEM_NULL if the SUNLinearSolver memory was NULL
- SUNLS_ILL_INPUT if SUNDIALS was not built with monitoring enabled

Notes This function is intended for users that wish to monitor the linear solver progress. By default, the file pointer is set to `stdout`.

SUNDIALS **must be built with the CMake option** `SUNDIALS_BUILD_WITH_MONITORING`, **to utilize this function.** See section [A.1.2](#) for more information.

F2003 Name FSUNLinSolSetInfoFile_SPTFQMR

SUNLinSolSetPrintLevel_SPTFQMR

Call `retval = SUNLinSolSetPrintLevel_SPTFQMR(NLS, print_level);`

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

Arguments `LS` (`SUNLinearSolver`) a `SUNNONLINSOL` object
`print_level` (`int`) flag indicating level of verbosity; must be one of:

- 0, no information is printed (default)
- 1, for each linear iteration the residual norm is printed

Return value The return value is

- SUNLS_SUCCESS if successful
- SUNLS_MEM_NULL if the SUNLinearSolver memory was NULL
- SUNLS_ILL_INPUT if SUNDIALS was not built with monitoring enabled, or the print level value was invalid

Notes This function is intended for users that wish to monitor the linear solver progress. By default, the print level is 0.

SUNDIALS **must be built with the CMake option** `SUNDIALS_BUILD_WITH_MONITORING`, **to utilize this function.** See section [A.1.2](#) for more information.

F2003 Name FSUNLinSolSetPrintLevel_SPTFQMR

9.18.3 SUNLinearSolver_SPTFQMR Fortran interfaces

The `SUNLINSOL_SPFGMR` module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The `fsunlinsol_sptfqmr_mod` FORTRAN module defines interfaces to all `SUNLINSOL_SPFGMR` C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function `SUNLinSol_SPTFQMR` is interfaced as `FSUNLinSol_SPTFQMR`.

The FORTRAN 2003 `SUNLINSOL_SPFGMR` interface module can be accessed with the `use` statement, i.e. `use fsunlinsol_sptfqmr_mod`, and linking to the library `libsundials_fsunlinsolsptfqmr_mod.lib` in addition to the C library. For details on where the library and module file `fsunlinsol_sptfqmr_mod.mod` are installed see [Appendix A](#). We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators *without* separately linking to the `libsundials_fsunlinsolsptfqmr_mod` library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL_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 SUNLINSOL_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 SUNLINSOL_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 SUNLINSOL_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 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_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 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_SPTFQMRSetPrecType for complete further documentation of this routine.

FSUNMASSPTFQMRSETPRECTYPE

Call `FSUNMASSPTFQMRSETPRECTYPE(pretype, ier)`

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

Arguments The arguments are identical to `FSUNPTFQMRSETPRECTYPE` 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_SPTFQMRSetPrecType` for complete further documentation of this routine.

FSUNPTFQMRSETMAXL

Call `FSUNPTFQMRSETMAXL(code, maxl, ier)`

Description The function `FSUNPTFQMRSETMAXL` 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 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_SPTFQMRSetMaxl` for complete further documentation of this routine.

FSUNMASSPTFQMRSETMAXL

Call `FSUNMASSPTFQMRSETMAXL(maxl, ier)`

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

Arguments The arguments are identical to `FSUNPTFQMRSETMAXL` 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_SPTFQMRSetMaxl` for complete further documentation of this routine.

9.18.4 SUNLinearSolver_SPTFQMR content

The `SUNLINSOL_SPTFQMR` module defines the *content* field of a `SUNLinearSolver` as the following structure:

```
struct _SUNLinearSolverContent_SPTFQMR {
    int maxl;
    int pretype;
    booleantype zeroguess;
    int numiters;
    realtype resnorm;
    int last_flag;
    ATimesFn ATimes;
    void* ATData;
    PSetupFn Psetup;
    PSolveFn Psolve;
    void* PData;
```

```

N_Vector s1;
N_Vector s2;
N_Vector r_star;
N_Vector q;
N_Vector d;
N_Vector v;
N_Vector p;
N_Vector *r;
N_Vector u;
N_Vector vtemp1;
N_Vector vtemp2;
N_Vector vtemp3;
int      print_level;
FILE*    info_file;
};

```

These entries of the *content* field contain the following information:

```

maxl      - number of TFQMR iterations to allow (default is 5),
pretype   - flag for type of preconditioning to employ (default is none),
numiters  - number of iterations from the most-recent solve,
resnorm   - final linear residual norm from the most-recent solve,
last_flag - last error return flag from an internal function,
ATimes    - function pointer to perform  $Av$  product,
ATData    - pointer to structure for ATimes,
Psetup    - function pointer to preconditioner setup routine,
Psolve    - function pointer to preconditioner solve routine,
PData     - pointer to structure for Psetup and Psolve,
s1, s2    - vector pointers for supplied scaling matrices (default is NULL),
r_star    - a NVECTOR which holds the initial scaled, preconditioned linear system residual,
q, d, v, p, u - NVECTORS used for workspace by the SPTFQMR algorithm,
r         - array of two NVECTORS used for workspace within the SPTFQMR algorithm,
vtemp1, vtemp2, vtemp3 - temporary vector storage.
print_level - controls the amount of information to be printed to the info file
info_file  - the file where all informative (non-error) messages will be directed

```

9.19 The SUNLinearSolver_PCG implementation

This section describes the SUNLINSOL implementaiton of the PCG (Preconditioned Conjugate Gradient [25]) iterative linear solver. The SUNLINSOL_PCG module is designed to be compatible with any NVECTOR implementation that supports a minimal subset of operations (N_VClone, N_VDotProd, N_VScale, N_VLinearSum, N_VProd, and N_VDestroy). Unlike the SPGMR and SPFGMR algorithms, PCG requires a fixed amount of memory that does not increase with the number of allowed iterations.

To access the SUNLINSOL_PCG module, include the header file `sunlinsol/sunlinsol_pcg.h`. We note that the SUNLINSOL_PCG module is accessible from SUNDIALS packages *without* separately linking to the `libsundials_sunlinsolpcg` module library.

9.19.1 SUNLinearSolver_PCG description

Unlike all of the other iterative linear solvers supplied with SUNDIALS, PCG should only be used on *symmetric* linear systems (e.g. mass matrix linear systems encountered in ARKODE). As a result, the explanation of the role of scaling and preconditioning matrices given in general must be modified in this scenario. The PCG algorithm solves a linear system $Ax = b$ where A is a symmetric ($A^T = A$), real-valued matrix. Preconditioning is allowed, and is applied in a symmetric fashion on both the right and left. Scaling is also allowed and is applied symmetrically. We denote the preconditioner and scaling matrices as follows:

- P is the preconditioner (assumed symmetric),
- S is a diagonal matrix of scale factors.

The matrices A and P are not required explicitly; only routines that provide A and P^{-1} as operators are required. The diagonal of the matrix S is held in a single NVECTOR, supplied by the user.

In this notation, PCG applies the underlying CG algorithm to the equivalent transformed system

$$\tilde{A}\tilde{x} = \tilde{b} \tag{9.3}$$

where

$$\begin{aligned} \tilde{A} &= SP^{-1}AP^{-1}S, \\ \tilde{b} &= SP^{-1}b, \\ \tilde{x} &= S^{-1}Px. \end{aligned} \tag{9.4}$$

The scaling matrix must be chosen so that the vectors $SP^{-1}b$ and $S^{-1}Px$ have dimensionless components.

The stopping test for the PCG iterations is on the L2 norm of the scaled preconditioned residual:

$$\begin{aligned} &\|\tilde{b} - \tilde{A}\tilde{x}\|_2 < \delta \\ \Leftrightarrow & \\ &\|SP^{-1}b - SP^{-1}Ax\|_2 < \delta \\ \Leftrightarrow & \\ &\|P^{-1}b - P^{-1}Ax\|_S < \delta \end{aligned}$$

where $\|v\|_S = \sqrt{v^T S^T S v}$, with an input tolerance δ .

This solver is constructed to perform the following operations:

- During construction all NVECTOR solver data is allocated, with vectors cloned from a template NVECTOR that is input, and default solver parameters are set.
- User-facing “set” routines may be called to modify default solver parameters.
- Additional “set” routines are called by the SUNDIALS solver that interfaces with SUNLINSOL_PCG to supply the ATimes, PSetup, and Psolve function pointers and **s** scaling vector.
- In the “initialize” call, the solver parameters are checked for validity.
- In the “setup” call, any non-NULL PSetup function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic PSetup function and the solver-specific routine (solver-supplied or user-supplied).
- In the “solve” call the PCG iteration is performed. This will include scaling and preconditioning if those options have been supplied.

9.19.2 SUNLinearSolver_PCG functions

The SUNLINSOL_PCG module provides the following user-callable constructor for creating a SUNLinearSolver object.

SUNLinSol_PCG	
Call	LS = SUNLinSol_PCG(y, pretype, max1);
Description	The function SUNLinSol_PCG creates and allocates memory for a PCG SUNLinearSolver object.
Arguments	<p>y (N_Vector) a template for cloning vectors needed within the solver</p> <p>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).</p> <p>max1 (int) the number of linear iterations to allow; values ≤ 0 will result in the default value (5).</p>
Return value	This returns a SUNLinearSolver object. If either y is incompatible then this routine will return NULL.
Notes	<p>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.</p> <p>Although some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL), PCG should <i>only</i> be used with these packages when the linear systems are known to be <i>symmetric</i>. 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.</p>
Deprecated Name	For backward compatibility, the wrapper function SUNPCG with identical input and output arguments is also provided.
F2003 Name	FSUNLinSol_PCG

The SUNLINSOL_PCG module defines implementations of all “iterative” linear solver operations listed in Sections 9.1.1 – 9.1.3:

- 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
- SUNLinSolSetZeroGuess_PCG – note the solver assumes a non-zero guess by default and the zero guess flag is reset to SUNFALSE after each call to SUNLinSolSolve_PCG.
- SUNLinSolSetup_PCG
- SUNLinSolSolve_PCG
- SUNLinSolNumIters_PCG
- SUNLinSolResNorm_PCG

- SUNLinSolResid_PCG
- SUNLinSolLastFlag_PCG
- SUNLinSolSpace_PCG
- SUNLinSolFree_PCG

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an ‘F’ to the function name.

The SUNLINSOL_PCG module also defines the following additional user-callable functions.

SUNLinSol_PCGSetPrecType

Call	<code>retval = SUNLinSol_PCGSetPrecType(LS, pretype);</code>
Description	The function <code>SUNLinSol_PCGSetPrecType</code> updates the flag indicating use of preconditioning in the <code>SUNLINSOL_PCG</code> object.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) the <code>SUNLINSOL_PCG</code> object to update <code>pretype</code> (<code>int</code>) flag indicating use of preconditioning, allowed values match those discussed in <code>SUNLinSol_PCG</code> .
Return value	This routine will return with one of the error codes <code>SUNLS_ILL_INPUT</code> (illegal <code>pretype</code>), <code>SUNLS_MEM_NULL</code> (<code>S</code> is <code>NULL</code>) or <code>SUNLS_SUCCESS</code> .
Deprecated Name	For backward compatibility, the wrapper function <code>SUNPCGSetPrecType</code> with identical input and output arguments is also provided.
F2003 Name	<code>FSUNLinSol_PCGSetPrecType</code>

SUNLinSol_PCGSetMax1

Call	<code>retval = SUNLinSol_PCGSetMax1(LS, max1);</code>
Description	The function <code>SUNLinSol_PCGSetMax1</code> updates the number of linear solver iterations to allow.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) the <code>SUNLINSOL_PCG</code> object to update <code>max1</code> (<code>int</code>) flag indicating the number of iterations to allow; values ≤ 0 will result in the default value (5)
Return value	This routine will return with one of the error codes <code>SUNLS_MEM_NULL</code> (<code>S</code> is <code>NULL</code>) or <code>SUNLS_SUCCESS</code> .
Deprecated Name	For backward compatibility, the wrapper function <code>SUNPCGSetMax1</code> with identical input and output arguments is also provided.
F2003 Name	<code>FSUNLinSol_PCGSetMax1</code>

SUNLinSolSetInfoFile_PCG

Call	<code>retval = SUNLinSolSetInfoFile_PCG(LS, info_file);</code>
Description	The function <code>SUNLinSolSetInfoFile_PCG</code> sets the output file where all informative (non-error) messages should be directed.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) a <code>SUNNONLINSOL</code> object <code>info_file</code> (<code>FILE*</code>) pointer to output file (<code>stdout</code> by default); a <code>NULL</code> input will disable output
Return value	The return value is

- `SUNLS_SUCCESS` if successful
- `SUNLS_MEM_NULL` if the `SUNLinearSolver` memory was `NULL`

- SUNLS_ILL_INPUT if SUNDIALS was not built with monitoring enabled

Notes This function is intended for users that wish to monitor the linear solver progress. By default, the file pointer is set to `stdout`.

SUNDIALS **must be built with the CMake option** `SUNDIALS_BUILD_WITH_MONITORING`, **to utilize this function.** See section [A.1.2](#) for more information.

F2003 Name `FSUNLinSolSetInfoFile_PCG`

`SUNLinSolSetPrintLevel_PCG`

Call `retval = SUNLinSolSetPrintLevel_PCG(NLS, print_level);`

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

Arguments `LS` (`SUNLinearSolver`) a `SUNNONLINSOL` object
`print_level` (`int`) flag indicating level of verbosity; must be one of:

- 0, no information is printed (default)
- 1, for each linear iteration the residual norm is printed

Return value The return value is

- `SUNLS_SUCCESS` if successful
- `SUNLS_MEM_NULL` if the `SUNLinearSolver` memory was `NULL`
- `SUNLS_ILL_INPUT` if SUNDIALS was not built with monitoring enabled, or the print level value was invalid

Notes This function is intended for users that wish to monitor the linear solver progress. By default, the print level is 0.

SUNDIALS **must be built with the CMake option** `SUNDIALS_BUILD_WITH_MONITORING`, **to utilize this function.** See section [A.1.2](#) for more information.

F2003 Name `FSUNLinSolSetPrintLevel_PCG`

9.19.3 SUNLinearSolver_PCG Fortran interfaces

The `SUNLINSOL_PCG` module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The `fsunlinsol_pcg_mod` FORTRAN module defines interfaces to all `SUNLINSOL_PCG` C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function `SUNLinSol_PCG` is interfaced as `FSUNLinSol_PCG`.

The FORTRAN 2003 `SUNLINSOL_PCG` interface module can be accessed with the `use` statement, i.e. `use fsunlinsol_pcg_mod`, and linking to the library `libsundials_fsunlinsolpcg_mod.lib` in addition to the C library. For details on where the library and module file `fsunlinsol_pcg_mod.mod` are installed see Appendix [A](#). We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators *without* separately linking to the `libsundials_fsunlinsolpcg_mod` library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 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 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_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 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_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 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_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 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.

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.

9.19.4 SUNLinearSolver_PCG content

The `SUNLINSOL_PCG` module defines the *content* field of a `SUNLinearSolver` as the following structure:

```
struct _SUNLinearSolverContent_PCG {
    int maxl;
    int pretype;
    booleantype zeroguess;
    int numiters;
    reatype resnorm;
    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;
    int print_level;
    FILE* info_file;
};
```

These entries of the *content* field contain the following information:

maxl - number of PCG iterations to allow (default is 5),
pretype - flag for use of preconditioning (default is none),
numiters - number of iterations from the most-recent solve,
resnorm - final linear residual norm from the most-recent solve,
last_flag - last error return flag from an internal function,
ATimes - function pointer to perform Av product,
ATData - pointer to structure for **ATimes**,
Psetup - function pointer to preconditioner setup routine,
Psolve - function pointer to preconditioner solve routine,
PData - pointer to structure for **Psetup** and **Psolve**,
s - vector pointer for supplied scaling matrix (default is NULL),
r - a NVECTOR which holds the preconditioned linear system residual,
p, z, Ap - NVECTORS used for workspace by the PCG algorithm.
print_level - controls the amount of information to be printed to the info file
info_file - the file where all informative (non-error) messages will be directed

9.20 SUNLinearSolver Examples

There are `SUNLinearSolver` examples that may be installed for each implementation; these make use of the functions in `test_sunlinsol.c`. These example functions show simple usage of the `SUNLinearSolver` family of functions. The inputs to the examples depend on the linear solver type, and are output to `stdout` if the example is run without the appropriate number of command-line arguments.

The following is a list of the example functions in `test_sunlinsol.c`:

- **Test_SUNLinSolGetType**: Verifies the returned solver type against the value that should be returned.
- **Test_SUNLinSolInitialize**: Verifies that `SUNLinSolInitialize` can be called and returns successfully.
- **Test_SUNLinSolSetup**: Verifies that `SUNLinSolSetup` can be called and returns successfully.
- **Test_SUNLinSolSolve**: Given a `SUNMATRIX` object A , `NVECTOR` objects x and b (where $Ax = b$) and a desired solution tolerance `tol`, this routine clones x into a new vector y , calls `SUNLinSolSolve` to fill y as the solution to $Ay = b$ (to the input tolerance), verifies that each entry in x and y match to within $10*tol$, and overwrites x with y prior to returning (in case the calling routine would like to investigate further).
- **Test_SUNLinSolSetATimes** (iterative solvers only): Verifies that `SUNLinSolSetATimes` can be called and returns successfully.
- **Test_SUNLinSolSetPreconditioner** (iterative solvers only): Verifies that `SUNLinSolSetPreconditioner` can be called and returns successfully.
- **Test_SUNLinSolSetScalingVectors** (iterative solvers only): Verifies that `SUNLinSolSetScalingVectors` can be called and returns successfully.
- **Test_SUNLinSolLastFlag**: Verifies that `SUNLinSolLastFlag` can be called, and outputs the result to `stdout`.
- **Test_SUNLinSolNumIters** (iterative solvers only): Verifies that `SUNLinSolNumIters` can be called, and outputs the result to `stdout`.

- `Test_SUNLinSolResNorm` (iterative solvers only): Verifies that `SUNLinSolResNorm` can be called, and that the result is non-negative.
- `Test_SUNLinSolResid` (iterative solvers only): Verifies that `SUNLinSolResid` can be called.
- `Test_SUNLinSolSpace` verifies that `SUNLinSolSpace` can be called, and outputs the results to `stdout`.

We'll note that these tests should be performed in a particular order. For either direct or iterative linear solvers, `Test_SUNLinSolInitialize` must be called before `Test_SUNLinSolSetup`, which must be called before `Test_SUNLinSolSolve`. Additionally, for iterative linear solvers `Test_SUNLinSolSetATimes`, `Test_SUNLinSolSetPreconditioner` and `Test_SUNLinSolSetScalingVectors` should be called before `Test_SUNLinSolInitialize`; similarly `Test_SUNLinSolNumIters`, `Test_SUNLinSolResNorm` and `Test_SUNLinSolResid` should be called after `Test_SUNLinSolSolve`. These are called in the appropriate order in all of the example problems.

Chapter 10

Description of the SUNMemory module

To support applications which leverage memory pools, or utilize a memory abstraction layer, SUNDIALS provides a set of utilities we will collectively refer to as the `SUNMemoryHelper` API. The goal of this API is to allow users to leverage operations defined by native SUNDIALS data structures while allowing the user to have finer-grained control of the memory management.

10.1 The SUNMemoryHelper API

This API consists of three new SUNDIALS types: `SUNMemoryType`, `SUNMemory`, and `SUNMemoryHelper`, which we now define.

The `SUNMemory` structure wraps a pointer to actual data. This structure is defined as

```
typedef struct _SUNMemory
{
    void*          ptr;
    SUNMemoryType type;
    booleantype   own;
} *SUNMemory;
```

The `SUNMemoryType` type is an enumeration that defines the four supported memory types:

```
typedef enum
{
    SUNMEMTYPE_HOST,      /* pageable memory accessible on the host */
    SUNMEMTYPE_PINNED,   /* page-locked memory accessible on the host */
    SUNMEMTYPE_DEVICE,   /* memory accessible from the device */
    SUNMEMTYPE_UVM       /* memory accessible from the host or device */
} SUNMemoryType;
```

Finally, the `SUNMemoryHelper` structure is defined as

```
struct _SUNMemoryHelper
{
    void*          content;
    SUNMemoryHelper_Ops ops;
} *SUNMemoryHelper;
```

where `SUNMemoryHelper_Ops` is defined as

```

typedef struct _SUNMemoryHelper_Ops
{
    /* operations that implementations are required to provide */
    int      (*alloc)(SUNMemoryHelper, SUNMemory* memptr, size_t mem_size, SUNMemoryType mem_type);
    int      (*dealloc)(SUNMemoryHelper, SUNMemory mem);
    int      (*copy)(SUNMemoryHelper, SUNMemory dst, SUNMemory src, size_t mem_size);

    /* operations that provide default implementations */
    int      (*copyasync)(SUNMemoryHelper, SUNMemory dst, SUNMemory src,
                          size_t mem_size, void* ctx);
    SUNMemoryHelper (*clone)(SUNMemoryHelper);
    int      (*destroy)(SUNMemoryHelper);
} *SUNMemoryHelper_Ops;

```

10.1.1 Implementation defined operations

The SUNMemory API also defines the following operations which do require a SUNMemoryHelper instance and **require** the implementation to define them:

SUNMemoryHelper_Alloc

Call `retval = SUNMemoryHelper_Alloc(helper, *memptr, mem_size, mem_type);`

Description Allocates a SUNMemory object whose ptr field is allocated for mem_size bytes and is of type mem_type. The new object will have ownership of ptr and will be deallocated when SUNMemoryHelper_Dealloc is called.

Arguments `helper` (SUNMemoryHelper) the SUNMemoryHelper object
`memptr` (SUNMemory*) pointer to the allocated SUNMemory
`mem_size` (size_t) the size in bytes of the ptr
`mem_type` (SUNMemoryType) the SUNMemoryType of the ptr

Return value An int flag indicating success (zero) or failure (non-zero).

SUNMemoryHelper_Dealloc

Call `retval = SUNMemoryHelper_Dealloc(helper, mem);`

Description Deallocates the mem->ptr field if it is owned by mem, and then deallocates the mem object.

Arguments `helper` (SUNMemoryHelper) the SUNMemoryHelper object
`mem` (SUNMemory) the SUNMemory object

Return value An int flag indicating success (zero) or failure (non-zero).

SUNMemoryHelper_Copy

Call `retval = SUNMemoryHelper_Copy(helper, dst, src, mem_size);`

Description Synchronously copies mem_size bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The helper object should use the memory types of dst and src to determine the appropriate transfer type necessary.

Arguments `helper` (SUNMemoryHelper) the SUNMemoryHelper object
`dst` (SUNMemory) the destination memory to copy to
`src` (SUNMemory) the source memory to copy from
`mem_size` (size_t) the number of bytes to copy

Return value An int flag indicating success (zero) or failure (non-zero).

10.1.2 Utility Functions

The SUNMemoryHelper API defines the following functions which do not require a SUNMemoryHelper instance:

SUNMemoryHelper_Alias

Call `mem2 = SUNMemoryHelper_Alias(mem1);`

Description Returns a SUNMemory object whose `ptr` field points to the same address as `mem1`. The new object *will not* have ownership of `ptr`, therefore, it will not free `ptr` when `SUNMemoryHelper_Dealloc` is called.

Arguments `mem1` (SUNMemory) a SUNMemory object

Return value A SUNMemory object.

SUNMemoryHelper_Wrap

Call `mem = SUNMemoryHelper_Wrap(ptr, mem_type);`

Description Returns a SUNMemory object whose `ptr` field points to the `ptr` argument passed to the function. The new object *will not* have ownership of `ptr`, therefore, it will not free `ptr` when `SUNMemoryHelper_Dealloc` is called.

Arguments `ptr` (SUNMemoryType) the data pointer to wrap in a SUNMemory object

`mem_type` (SUNMemoryType) the SUNMemoryType of the `ptr`

Return value A SUNMemory object.

SUNMemoryHelper_NewEmpty

Call `helper = SUNMemoryHelper_NewEmpty();`

Description Returns an empty SUNMemoryHelper. This is useful for building custom SUNMemoryHelper implementations.

Arguments

Return value A SUNMemoryHelper object.

SUNMemoryHelper_CopyOps

Call `retval = SUNMemoryHelper_CopyOps(src, dst);`

Description Copies the `ops` field of `src` to the `ops` field of `dst`. This is useful for building custom SUNMemoryHelper implementations.

Arguments `src` (SUNMemoryHelper) the object to copy from

`dst` (SUNMemoryHelper) the object to copy to

Return value An `int` flag indicating success (zero) or failure (non-zero).

10.1.3 Implementation overridable operations with defaults

In addition, the SUNMemoryHelper API defines the following *optionally overridable* operations which do require a SUNMemoryHelper instance:

SUNMemoryHelper_CopyAsync

Call `retval = SUNMemoryHelper_CopyAsync(helper, dst, src, mem_size, ctx);`

Description Asynchronously copies `mem_size` bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The `helper` object should use the memory types of `dst` and `src` to determine the appropriate transfer type necessary. The `ctx` argument is used when a different execution “stream” needs to be provided to perform the copy in, e.g. with CUDA this would be a `cudaStream_t`.

Arguments `helper` (`SUNMemoryHelper`) the `SUNMemoryHelper` object
`dst` (`SUNMemory`) the destination memory to copy to
`src` (`SUNMemory`) the source memory to copy from
`mem_size` (`size_t`) the number of bytes to copy
`ctx` (`void *`) typically a handle for an object representing an alternate execution stream, but it can be any implementation specific data

Return value An `int` flag indicating success (zero) or failure (non-zero).



Notes If this operation is not defined by the implementation, then `SUNMemoryHelper_Copy` will be used.

SUNMemoryHelper_Clone

Call `helper2 = SUNMemoryHelper_Clone(helper);`

Description Clones the `SUNMemoryHelper` object itself.

Arguments `helper` (`SUNMemoryHelper`) the `SUNMemoryHelper` object to clone

Return value A `SUNMemoryHelper` object.



Notes If this operation is not defined by the implementation, then the default clone will only copy the `SUNMemoryHelper_Ops` structure stored in `helper->ops`, and not the `helper->content` field.

SUNMemoryHelper_Destroy

Call `retval = SUNMemoryHelper_Destroy(helper);`

Description Destroys (frees) the `SUNMemoryHelper` object itself.

Arguments `helper` (`SUNMemoryHelper`) the `SUNMemoryHelper` object to destroy

Return value An `int` flag indicating success (zero) or failure (non-zero).



Notes If this operation is not defined by the implementation, then the default destroy will only free the `helper->ops` field and the `helper` itself. The `helper->content` field will not be freed.

10.1.4 Implementing a custom `SUNMemoryHelper`

A particular implementation of the `SUNMemoryHelper` API must:

- Define and implement the required operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one `SUNMemoryHelper` module in the same code.
- Optionally, specify the `content` field of `SUNMemoryHelper`.
- Optionally, define and implement additional user-callable routines acting on the newly defined `SUNMemoryHelper`.

An example of a custom `SUNMemoryHelper` is given in `examples/utilities/custom_memory_helper.h`.

10.2 The SUNMemoryHelper_Cuda implementation

The `SUNMemoryHelper_Cuda` module is an implementation of the `SUNMemoryHelper` API that interfaces to the NVIDIA CUDA [5] library. The implementation defines the constructor

SUNMemoryHelper_Cuda

Call `helper = SUNMemoryHelper_Cuda();`

Description Allocates and returns a `SUNMemoryHelper` object for handling CUDA memory.

Arguments None

Return value A `SUNMemoryHelper` object if successful, or `NULL` if not.

10.2.1 SUNMemoryHelper API functions

The implementation provides the following operations defined by the `SUNMemoryHelper` API:

SUNMemoryHelper_Alloc_Cuda

Call `retval = SUNMemoryHelper_Alloc_Cuda(helper, *memptr, mem_size, mem_type);`

Description Allocates a `SUNMemory` object whose `ptr` field is allocated for `mem_size` bytes and is of type `mem_type`. The new object will have ownership of `ptr` and will be deallocated when `SUNMemoryHelper_Dealloc` is called.

The `SUNMemoryType` supported are

- `SUNMEMTYPE_HOST` – memory is allocated with a call to `malloc`
- `SUNMEMTYPE_PINNED` – memory is allocated with a call to `cudaMallocHost`
- `SUNMEMTYPE_DEVICE` – memory is allocated with a call to `cudaMalloc`
- `SUNMEMTYPE_UVM` – memory is allocated with a call to `cudaMallocManaged`

Arguments `helper` (`SUNMemoryHelper`) the `SUNMemoryHelper` object
`memptr` (`SUNMemory*`) pointer to the allocated `SUNMemory`
`mem_size` (`size_t`) the size in bytes of the `ptr`
`mem_type` (`SUNMemoryType`) the `SUNMemoryType` of the `ptr`

Return value An `int` flag indicating success (zero) or failure (non-zero).

SUNMemoryHelper_Dealloc_Cuda

Call `retval = SUNMemoryHelper_Dealloc_Cuda(helper, mem);`

Description Deallocates the `mem->ptr` field if it is owned by `mem`, and then deallocates the `mem` object.

Arguments `helper` (`SUNMemoryHelper`) the `SUNMemoryHelper` object
`mem` (`SUNMemory`) the `SUNMemory` object

Return value An `int` flag indicating success (zero) or failure (non-zero).

SUNMemoryHelper_Copy_Cuda

Call `retval = SUNMemoryHelper_Copy_Cuda(helper, dst, src, mem_size);`

Description Synchronously copies `mem_size` bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The `helper` object will use the memory types of `dst` and `src` to determine the appropriate transfer type necessary.

Arguments This operation uses `cudaMemcpy` underneath.

Return value `helper` (SUNMemoryHelper) the SUNMemoryHelper object
`dst` (SUNMemory) the destination memory to copy to
`src` (SUNMemory) the source memory to copy from
`mem_size` (size_t) the number of bytes to copy

Notes An int flag indicating success (zero) or failure (non-zero).

SUNMemoryHelper_CopyAsync_Cuda

Call `retval = SUNMemoryHelper_CopyAsync_Cuda(helper, dst, src, mem_size, ctx);`

Description Asynchronously copies `mem_size` bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The `helper` object will use the memory types of `dst` and `src` to determine the appropriate transfer type necessary.

Arguments This operation uses `cudaMemcpyAsync` underneath.

Return value `helper` (SUNMemoryHelper) the SUNMemoryHelper object
`dst` (SUNMemory) the destination memory to copy to
`src` (SUNMemory) the source memory to copy from
`mem_size` (size_t) the number of bytes to copy
`ctx` (void *) the `cudaStream_t` handle for the stream that the copy will be performed on

Notes An int flag indicating success (zero) or failure (non-zero).

10.3 The SUNMemoryHelper_Hip implementation

The `SUNMemoryHelper_Hip` module is an implementation of the `SUNMemoryHelper` API that interfaces to the AMD ROCm HIP library. The implementation defines the constructor

SUNMemoryHelper_Hip

Call `helper = SUNMemoryHelper_Hip();`

Description Allocates and returns a `SUNMemoryHelper` object for handling HIP memory.

Arguments None

Return value A `SUNMemoryHelper` object if successful, or `NULL` if not.

10.3.1 SUNMemoryHelper API functions

The implementation provides the following operations defined by the `SUNMemoryHelper` API:

SUNMemoryHelper_Alloc_Hip

Call `retval = SUNMemoryHelper_Alloc_Hip(helper, *memptr, mem_size, mem_type);`

Description Allocates a `SUNMemory` object whose `ptr` field is allocated for `mem_size` bytes and is of type `mem_type`. The new object will have ownership of `ptr` and will be deallocated when `SUNMemoryHelper_Dealloc` is called.

The `SUNMemoryType` supported are

- `SUNMEMTYPE_HOST` – memory is allocated with a call to `malloc`
- `SUNMEMTYPE_PINNED` – memory is allocated with a call to `hipMallocHost`
- `SUNMEMTYPE_DEVICE` – memory is allocated with a call to `hipMalloc`

- `SUNMEMTYPE_UVM` – memory is allocated with a call to `hipMallocManaged`

Arguments `helper` (`SUNMemoryHelper`) the `SUNMemoryHelper` object
`memptr` (`SUNMemory*`) pointer to the allocated `SUNMemory`
`mem_size` (`size_t`) the size in bytes of the `ptr`
`mem_type` (`SUNMemoryType`) the `SUNMemoryType` of the `ptr`

Return value An `int` flag indicating success (zero) or failure (non-zero).

`SUNMemoryHelper_Dealloc_Hip`

Call `retval = SUNMemoryHelper_Dealloc_Hip(helper, mem);`

Description Deallocates the `mem->ptr` field if it is owned by `mem`, and then deallocates the `mem` object.

Arguments `helper` (`SUNMemoryHelper`) the `SUNMemoryHelper` object
`mem` (`SUNMemory`) the `SUNMemory` object

Return value An `int` flag indicating success (zero) or failure (non-zero).

`SUNMemoryHelper_Copy_Hip`

Call `retval = SUNMemoryHelper_Copy_Hip(helper, dst, src, mem_size);`

Description Synchronously copies `mem_size` bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The `helper` object will use the memory types of `dst` and `src` to determine the appropriate transfer type necessary.

Arguments This operation uses `hipMemcpy` underneath.

Return value `helper` (`SUNMemoryHelper`) the `SUNMemoryHelper` object
`dst` (`SUNMemory`) the destination memory to copy to
`src` (`SUNMemory`) the source memory to copy from
`mem_size` (`size_t`) the number of bytes to copy

Notes An `int` flag indicating success (zero) or failure (non-zero).

`SUNMemoryHelper_CopyAsync_Hip`

Call `retval = SUNMemoryHelper_CopyAsync_Hip(helper, dst, src, mem_size, ctx);`

Description Asynchronously copies `mem_size` bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The `helper` object will use the memory types of `dst` and `src` to determine the appropriate transfer type necessary.

Arguments This operation uses `hipMemcpyAsync` underneath.

Return value `helper` (`SUNMemoryHelper`) the `SUNMemoryHelper` object
`dst` (`SUNMemory`) the destination memory to copy to
`src` (`SUNMemory`) the source memory to copy from
`mem_size` (`size_t`) the number of bytes to copy
`ctx` (`void *`) the `hipStream_t` handle for the stream that the copy will be performed on

Notes An `int` flag indicating success (zero) or failure (non-zero).

10.4 The `SUNMemoryHelper_Sycl` implementation

The `SUNMemoryHelper_Sycl` module is an implementation of the `SUNMemoryHelper` API that interfaces to the SYCL abstraction layer. The implementation defines the constructor

SUNMemoryHelper_Sycl

Call `helper = SUNMemoryHelper_Sycl(Q);`

Description Allocates and returns a `SUNMemoryHelper` object for handling SYCL memory.

Arguments `Q (sycl::queue)` the queue to use for memory operations

Return value A `SUNMemoryHelper` object if successful, or `NULL` if not.

10.4.1 SUNMemoryHelper API functions

The implementation provides the following operations defined by the `SUNMemoryHelper` API:

SUNMemoryHelper_Alloc_Sycl

Call `retval = SUNMemoryHelper_Alloc_Sycl(helper, *memptr, mem_size, mem_type);`

Description Allocates a `SUNMemory` object whose `ptr` field is allocated for `mem_size` bytes and is of type `mem_type`. The new object will have ownership of `ptr` and will be deallocated when `SUNMemoryHelper_Dealloc` is called.

The `SUNMemoryType` supported are

- `SUNMEMTYPE_HOST` – memory is allocated with a call to `malloc`
- `SUNMEMTYPE_PINNED` – memory is allocated with a call to `sycl::malloc_host`
- `SUNMEMTYPE_DEVICE` – memory is allocated with a call to `sycl::malloc_device`
- `SUNMEMTYPE_UVM` – memory is allocated with a call to `sycl::malloc_shared`

Arguments `helper (SUNMemoryHelper)` the `SUNMemoryHelper` object
`memptr (SUNMemory*)` pointer to the allocated `SUNMemory`
`mem_size (size_t)` the size in bytes of the `ptr`
`mem_type (SUNMemoryType)` the `SUNMemoryType` of the `ptr`

Return value An `int` flag indicating success (zero) or failure (non-zero).

SUNMemoryHelper_Dealloc_Sycl

Call `retval = SUNMemoryHelper_Dealloc_Sycl(helper, mem);`

Description Deallocates the `mem->ptr` field if it is owned by `mem`, and then deallocates the `mem` object.

Arguments `helper (SUNMemoryHelper)` the `SUNMemoryHelper` object
`mem (SUNMemory)` the `SUNMemory` object

Return value An `int` flag indicating success (zero) or failure (non-zero).

SUNMemoryHelper_Copy_Sycl

Call `retval = SUNMemoryHelper_Copy_Sycl(helper, dst, src, mem_size);`

Description Synchronously copies `mem_size` bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The `helper` object will use the memory types of `dst` and `src` to determine the appropriate transfer type necessary.

Arguments This operation uses `syclMemcpy` underneath.

Return value `helper (SUNMemoryHelper)` the `SUNMemoryHelper` object
`dst (SUNMemory)` the destination memory to copy to
`src (SUNMemory)` the source memory to copy from
`mem_size (size_t)` the number of bytes to copy

Notes An `int` flag indicating success (zero) or failure (non-zero).

SUNMemoryHelper_CopyAsync_Sycl

Call `retval = SUNMemoryHelper_CopyAsync_Sycl(helper, dst, src, mem_size, ctx);`

Description Asynchronously copies `mem_size` bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The `helper` object will use the memory types of `dst` and `src` to determine the appropriate transfer type necessary.

Arguments This operation uses `syclMemcpyAsync` underneath.

Return value `helper` (`SUNMemoryHelper`) the `SUNMemoryHelper` object
`dst` (`SUNMemory`) the destination memory to copy to
`src` (`SUNMemory`) the source memory to copy from
`mem_size` (`size_t`) the number of bytes to copy
`ctx` (`void *`) is unused in this function

Notes An `int` flag indicating success (zero) or failure (non-zero).

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/CMAKE_INSTALL_LIBDIR`, with *instdir* and `CMAKE_INSTALL_LIBDIR` 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>. 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:

```
% 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

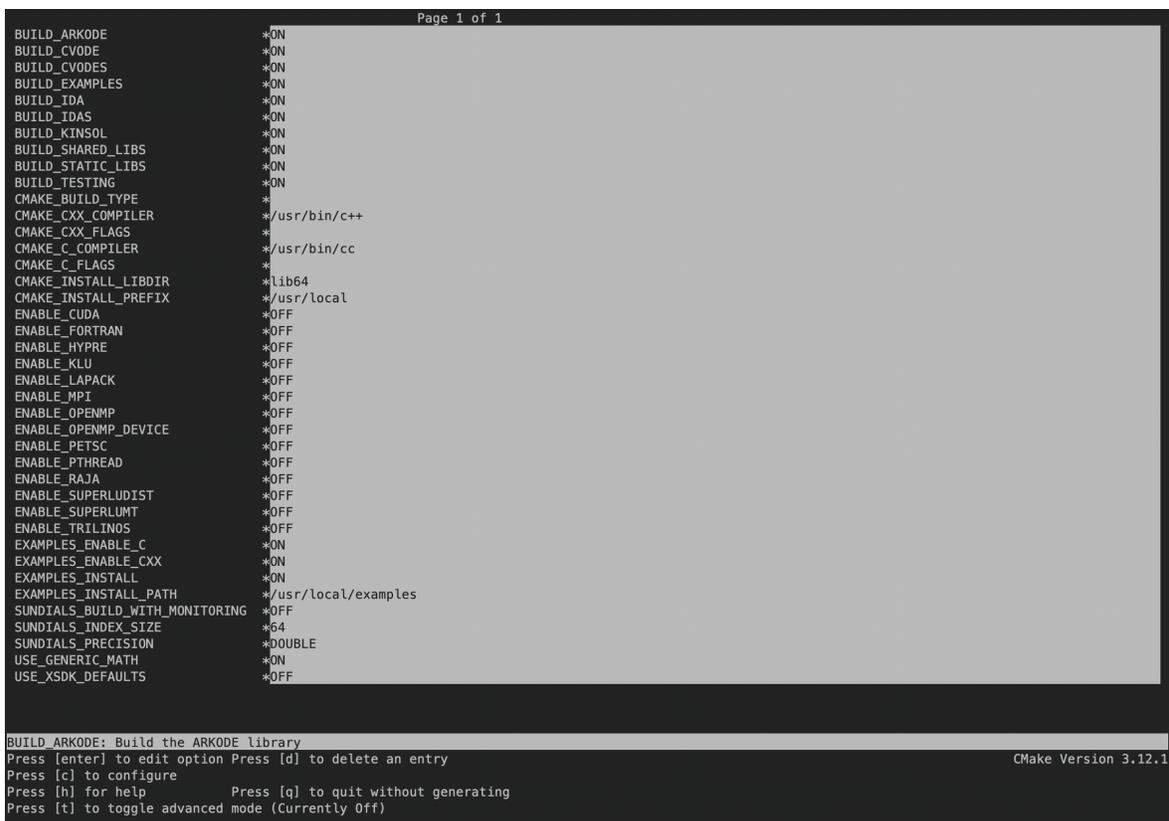
– For file and directories, the <tab> key can be used to complete

- 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.



```

Page 1 of 1
BUILD_ARKODE          *ON
BUILD_CVOICE         *ON
BUILD_CVOIDS         *ON
BUILD_EXAMPLES       *ON
BUILD_IDA            *ON
BUILD_IDAS           *ON
BUILD_KINSOL         *ON
BUILD_SHARED_LIBS    *ON
BUILD_STATIC_LIBS    *ON
BUILD_TESTING        *ON
CMAKE_BUILD_TYPE     *
CMAKE_CXX_COMPILER    */usr/bin/c++
CMAKE_CXX_FLAGS       *
CMAKE_C_COMPILER      */usr/bin/cc
CMAKE_C_FLAGS         *
CMAKE_INSTALL_LIBDIR  */lib64
CMAKE_INSTALL_PREFIX  */usr/local
ENABLE_CUDA          *OFF
ENABLE_FORTRAN       *OFF
ENABLE_HYPRE         *OFF
ENABLE_KLU           *OFF
ENABLE_LAPACK        *OFF
ENABLE_MPI           *OFF
ENABLE_OPENMP        *OFF
ENABLE_OPENMP_DEVICE *OFF
ENABLE_PETSC         *OFF
ENABLE_PTHREAD       *OFF
ENABLE_RAJA          *OFF
ENABLE_SUPERLUDIST   *OFF
ENABLE_SUPERLUMT     *OFF
ENABLE_TRILINOS      *OFF
EXAMPLES_ENABLE_C    *ON
EXAMPLES_ENABLE_CXX  *ON
EXAMPLES_INSTALL     *ON
EXAMPLES_INSTALL_PATH */usr/local/examples
SUNDIALS_BUILD_WITH_MONITORING *OFF
SUNDIALS_INDEX_SIZE  *64
SUNDIALS_PRECISION   *DOUBLE
USE_GENERIC_MATH     *ON
USE_XSDK_DEFAULTS    *OFF

BUILD_ARKODE: Build the ARKODE library
Press [enter] to edit option Press [d] to delete an entry
Press [c] to configure
Press [h] for help          Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
CMake Version 3.12.1

```

Figure A.1: Default configuration screen. Note: Initial screen is empty. To get this default configuration, press 'c' repeatedly (accepting default values denoted with asterisk) until the 'g' option is available.

The default *instdir* for both SUNDIALS and corresponding examples can be changed by setting the `CMAKE_INSTALL_PREFIX` and the `EXAMPLES_INSTALL_PATH` as shown in figure A.2.

Pressing the (g key) will generate makefiles including all dependencies and all rules to build SUNDIALS on this system. Back at the command prompt, you can now run:

```
% make
```

To install SUNDIALS in the installation directory specified in the configuration, simply run:

```
% make install
```

```

Page 1 of 1
BUILD_ARKODE *ON
BUILD_CVOICE *ON
BUILD_CVODES *ON
BUILD_EXAMPLES *ON
BUILD_IDA *ON
BUILD_IDAS *ON
BUILD_KINSOL *ON
BUILD_SHARED_LIBS *ON
BUILD_STATIC_LIBS *ON
BUILD_TESTING *ON
CMAKE_BUILD_TYPE *
CMAKE_CXX_COMPILER */usr/bin/c++
CMAKE_CXX_FLAGS *
CMAKE_C_COMPILER */usr/bin/cc
CMAKE_C_FLAGS *
CMAKE_INSTALL_LIBDIR *lib64
CMAKE_INSTALL_PREFIX */usr/casc/sundials/instdir
ENABLE_CUDA *OFF
ENABLE_FORTRAN *OFF
ENABLE_HYPRE *OFF
ENABLE_KLU *OFF
ENABLE_LAPACK *OFF
ENABLE_MPI *OFF
ENABLE_OPENMP *OFF
ENABLE_OPENMP_DEVICE *OFF
ENABLE_PETSC *OFF
ENABLE_PTHREAD *OFF
ENABLE_RAJA *OFF
ENABLE_SUPERLU_DIST *OFF
ENABLE_SUPERLU_MT *OFF
ENABLE_TRILINOS *OFF
EXAMPLES_ENABLE_C *ON
EXAMPLES_ENABLE_CXX *ON
EXAMPLES_INSTALL *ON
EXAMPLES_INSTALL_PATH */usr/casc/sundials/instdir/examples
SUNDIALS_BUILD_WITH_MONITORING *OFF
SUNDIALS_INDEX_SIZE *64
SUNDIALS_PRECISION *DOUBLE
USE_GENERIC_MATH *ON
USE_XSDK_DEFAULTS *OFF

EXAMPLES_INSTALL_PATH: Output directory for installing example files
Press [enter] to edit option Press [d] to delete an entry
Press [c] to configure
Press [h] for help Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
CMake Version 3.12.1

```

Figure A.2: Changing the *instdir* for SUNDIALS and corresponding examples

Building from the command line

Using CMake from the command line is simply a matter of specifying CMake variable settings with the `cmake` command. The following will build the default configuration:

```

% cmake -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> ../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 provided below. Note that the default values shown are for a typical configuration on a Linux system and are provided as illustration only.

BUILD_ARKODE - Build the ARKODE library
Default: ON

BUILD_CVOICE - Build the CVOICE library
Default: ON

BUILD_CVODES - Build the CVODES library
Default: ON

BUILD_IDA - Build the IDA library

Default: ON

BUILD_IDAS - Build the IDAS library

Default: ON

BUILD_KINSOL - Build the KINSOL library

Default: ON

BUILD_SHARED_LIBS - Build shared libraries

Default: ON

BUILD_STATIC_LIBS - Build static libraries

Default: ON

CMAKE_BUILD_TYPE - Choose the type of build, options are: `None` (CMAKE_C_FLAGS used), `Debug`, `Release`, `RelWithDebInfo`, and `MinSizeRel`

Default:

Note: Specifying a build type will trigger the corresponding build type specific compiler flag options below which will be appended to the flags set by CMAKE_<language>_FLAGS.

CMAKE_C_COMPILER - C compiler

Default: /usr/bin/cc

CMAKE_C_FLAGS - Flags for C compiler

Default:

CMAKE_C_FLAGS_DEBUG - Flags used by the C compiler during debug builds

Default: -g

CMAKE_C_FLAGS_MINSIZEREL - Flags used by the C compiler during release minsize builds

Default: -Os -DNDEBUG

CMAKE_C_FLAGS_RELEASE - Flags used by the C compiler during release builds

Default: -O3 -DNDEBUG

CMAKE_CXX_COMPILER - C++ compiler

Default: /usr/bin/c++

Note: A C++ compiler (and all related options) are only triggered if C++ examples are enabled (EXAMPLES_ENABLE_CXX is ON). All SUNDIALS solvers can be used from C++ applications by default without setting any additional configuration options.

CMAKE_CXX_FLAGS - Flags for C++ compiler

Default:

CMAKE_CXX_FLAGS_DEBUG - Flags used by the C++ compiler during debug builds

Default: -g

CMAKE_CXX_FLAGS_MINSIZEREL - Flags used by the C++ compiler during release minsize builds

Default: -Os -DNDEBUG

CMAKE_CXX_FLAGS_RELEASE - Flags used by the C++ compiler during release builds

Default: -O3 -DNDEBUG

CMAKE_CXX_STANDARD - The C++ standard to build C++ parts of SUNDIALS with.

Default: 11

Note: Options are 98, 11, 14, 17, 20. This option is on used when a C++ compiler is required.

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 LAPACK support is enabled (**ENABLE_LAPACK** is ON).

CMAKE_Fortran_FLAGS - Flags for Fortran compiler

Default:

CMAKE_Fortran_FLAGS_DEBUG - Flags used by the Fortran compiler during debug builds

Default: -g

CMAKE_Fortran_FLAGS_MINSIZEREL - Flags used by the Fortran compiler during release minsize builds

Default: -Os

CMAKE_Fortran_FLAGS_RELEASE - Flags used by the Fortran compiler during release builds

Default: -O3

CMAKE_INSTALL_PREFIX - Install path prefix, prepended onto install directories

Default: /usr/local

Note: The user must have write access to the location specified through this option. Exported SUNDIALS header files and libraries will be installed under subdirectories **include** and **CMAKE_INSTALL_LIBDIR** of **CMAKE_INSTALL_PREFIX**, respectively.

CMAKE_INSTALL_LIBDIR - Library installation directory

Default:

Note: This is the directory within **CMAKE_INSTALL_PREFIX** that the SUNDIALS libraries will be installed under. The default is automatically set based on the operating system using the GNUInstallDirs CMake module.

Fortran_INSTALL_MODDIR - Fortran module installation directory

Default: fortran

ENABLE_CUDA - Build the SUNDIALS CUDA modules.

Default: OFF

CUDA_ARCH - Specifies the CUDA architecture to compile for.

Default: sm_30

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 unless **ENABLE_TRILINOS** is ON.

EXAMPLES_ENABLE_F77 - Build the SUNDIALS Fortran77 examples

Default: ON (if **F77_INTERFACE_ENABLE** is ON)

EXAMPLES_ENABLE_F90 - Build the SUNDIALS Fortran90 examples

Default: ON (if **F77_INTERFACE_ENABLE** is ON)

EXAMPLES_ENABLE_F2003 - Build the SUNDIALS Fortran2003 examples

Default: ON (if **BUILD_FORTRAN_MODULE_INTERFACE** is ON)

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**.

F77_INTERFACE_ENABLE - Enable Fortran-C support via the Fortran 77 interfaces

Default: OFF

BUILD_FORTRAN_MODULE_INTERFACE - Enable Fortran-C support via the Fortran 2003 interfaces

Default: OFF

ENABLE_HYPRE - 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

ENABLE_KLU - 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

ENABLE_LAPACK - Enable LAPACK support

Default: OFF

Note: Setting this option to ON will trigger additional CMake options. See additional information on building with LAPACK enabled in [A.1.4](#).

LAPACK_LIBRARIES - LAPACK (and BLAS) libraries

Default: /usr/lib/liblapack.so;/usr/lib/libblas.so

Note: CMake will search for libraries in your **LD_LIBRARY_PATH** prior to searching default system paths.

ENABLE_MAGMA - Enable MAGMA support.

Default: OFF

Note: Setting this option to ON will trigger additional options related to MAGMA.

MAGMA_DIR - Path to the root of a MAGMA installation.

Default: none

SUNDIALS_MAGMA_BACKENDS - Which MAGMA backend to use under the SUNDIALS MAGMA interface.

Default: CUDA

`ENABLE_MPI` - Enable MPI support. This will build the parallel `NVECTOR` and the MPI-aware version of the ManyVector library.

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 (`ENABLE_MPI` is ON) and C++ examples are enabled (`EXAMPLES_ENABLE_CXX` is ON). All SUNDIALS solvers can be used from C++ MPI applications by default without setting any additional configuration options other than `ENABLE_MPI`.

`MPI_Fortran_COMPILER` - `mpif77` or `mpif90` program

Default:

Note: This option is triggered only if MPI is enabled (`ENABLE_MPI` is ON) and Fortran-C support is enabled (`F77_INTERFACE_ENABLE` or `BUILD_FORTRAN_MODULE_INTERFACE` is ON).

`MPIEXEC_EXECUTABLE` - Specify the executable for running MPI programs

Default: `mpirun`

Note: This option is triggered only if MPI is enabled (`ENABLE_MPI` is ON).

`ENABLE_ONEMKL` - Enable oneMKL support.

Default: OFF

`ENABLE_OPENMP` - Enable OpenMP support (build the OpenMP `NVECTOR`).

Default: OFF

`OPENMP_DEVICE_ENABLE` - Enable OpenMP device offloading (build the `OpenMPDEV` `nvector`) if supported by the provided compiler.

Default: OFF

`OPENMP_DEVICE_WORKS` - **advanced option** - Skip the check done to see if the OpenMP provided by the compiler supports OpenMP device offloading.

Default: OFF

`ENABLE_PETSC` - Enable PETSc support

Default: OFF

Note: See additional information on building with PETSc enabled in ??.

`PETSC_DIR` - Path to PETSc installation

Default:

`PETSC_LIBRARIES` - **advanced option** - Semi-colon separated list of PETSc link libraries. Unless provided by the user, this is autopopulated based on the PETSc installation found in `PETSC_DIR`.

Default:

`PETSC_INCLUDES` - **advanced option** - Semi-colon separated list of PETSc include directories. Unless provided by the user, this is autopopulated based on the PETSc installation found in `PETSC_DIR`.

Default:

`ENABLE_PTHREAD` - Enable Pthreads support (build the Pthreads `NVECTOR`).

Default: OFF

`ENABLE_RAJA` - Enable RAJA support.

Default: OFF

Note: You need to enable CUDA, HIP, or SYCL in order to build the RAJA vector module.

`SUNDIALS_RAJA_BACKENDS` - If building SUNDIALS with RAJA support, this sets the RAJA backend to target. Values supported are CUDA, HIP, or SYCL.

Default: CUDA

`ENABLE_SUPERLUDIST` - Enable SuperLU_DIST support

Default: OFF

Note: See additional information on building with SuperLU_DIST enabled in [A.1.4](#).

`SUPERLUDIST_INCLUDE_DIR` - Path to SuperLU_DIST header files (typically SRC directory)

`SUPERLUDIST_LIBRARY_DIR` - Path to SuperLU_DIST installed library files

`SUPERLUDIST_LIBRARIES` - Semi-colon separated list of libraries needed for SuperLU_DIST

`SUPERLUDIST_OpenMP` - Enable SUNDIALS support for SuperLU_DIST built with OpenMP

Default: OFF

Note: SuperLU_DIST must be built with OpenMP support for this option to function properly. Additionally the environment variable `OMP_NUM_THREADS` must be set to the desired number of threads.

`ENABLE_SUPERLUMT` - Enable SUPERLUMT support

Default: OFF

Note: See additional information on building with SUPERLUMT 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_LIBRARIES` - Semi-colon separated list of libraries needed for SuperLU_MT

`SUPERLUMT_THREAD_TYPE` - Must be set to Pthread or OpenMP

Default: Pthread

`ENABLE_SYCL` - Enable SYCL support.

Default: OFF

Note: At present the only supported SYCL compiler is the DPC++ (Intel oneAPI) compiler. CMake does not currently support autodetection of SYCL compilers and `CMAKE_CXX_COMPILER` must be set to a valid SYCL compiler i.e., `dpccpp` in order to build with SYCL support.

`ENABLE_TRILINOS` - Enable Trilinos support (build the Tpetra NVECTOR).

Default: OFF

`Trilinos_DIR` - Path to the Trilinos install directory.

Default:

`TRILINOS_INTERFACE_C_COMPILER` - **advanced option** - Set the C compiler for building the Trilinos interface (i.e., `NVECTOR_TRILINOS` and the examples that use it).

Default: The C compiler exported from the found Trilinos installation if `USE_XSDK_DEFAULTS=OFF`. `CMAKE_C_COMPILER` or `MPI_C_COMPILER` if `USE_XSDK_DEFAULTS=ON`.

Note: It is recommended to use the same compiler that was used to build the Trilinos library.

`TRILINOS_INTERFACE_C_COMPILER_FLAGS` - **advanced option** - Set the C compiler flags for Trilinos interface (i.e., `NVECTOR_TRILINOS` and the examples that use it).

Default: The C compiler flags exported from the found Trilinos installation if `USE_XSDK_DEFAULTS=OFF`. `CMAKE_C_FLAGS` if `USE_XSDK_DEFAULTS=ON`.

Note: It is recommended to use the same flags that were used to build the Trilinos library.

TRILINOS_INTERFACE_CXX_COMPILER - **advanced option** - Set the C++ compiler for building Trilinos interface (i.e., `NVECTOR_TRILINOS` and the examples that use it).

Default: The C++ compiler exported from the found Trilinos installation if `USE_XSDK_DEFAULTS=OFF`. `CMAKE_CXX_COMPILER` or `MPI_CXX_COMPILER` if `USE_XSDK_DEFAULTS=ON`.

Note: It is recommended to use the same compiler that was used to build the Trilinos library.

TRILINOS_INTERFACE_CXX_COMPILER_FLAGS - **advanced option** - Set the C++ compiler flags for Trilinos interface (i.e., `NVECTOR_TRILINOS` and the examples that use it).

Default: The C++ compiler flags exported from the found Trilinos installation if `USE_XSDK_DEFAULTS=OFF`. `CMAKE_CXX_FLAGS` if `USE_XSDK_DEFAULTS=ON`.

Note: It is recommended to use the same flags that were used to build the Trilinos library.

SUNDIALS_BUILD_WITH_MONITORING - Build SUNDIALS with capabilities for fine-grained monitoring of solver progress and statistics. This is primarily useful for debugging.

Default: OFF

Note: Building with monitoring may result in minor performance degradation even if monitoring is not utilized.

SUNDIALS_BUILD_PACKAGE_FUSED_KERNELS - Build specialized fused kernels inside `CVODE`.

Default: OFF

Note: This option is currently only available when building with `CUDA_ENABLE = ON`. Building with fused kernels requires linking to either `libsundials_cvode_fused_cuda.lib` or `libsundials_cvode_fused_stub.lib` where the latter provides CPU-only placeholders for the fused routines, in addition to `libsundials_cvode.lib`.

CMAKE_CXX_STANDARD - The C++ standard to build C++ parts of SUNDIALS with.

Default: 11

Note: Options are 99, 11, 14, 17. This option only used when a C++ compiler is required.

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 option** - 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`

`SUNDIALS_INSTALL_CMAKEDIR` - Installation directory for the SUNDIALS cmake files (relative to `CMAKE_INSTALL_PREFIX`).
 Default: `CMAKE_INSTALL_PREFIX/cmake/sundials`

`USE_GENERIC_MATH` - Use generic (stdc) math libraries
 Default: `ON`

`USE_XSDK_DEFAULTS` - Enable xSDK (see [for more information](#)) default configuration settings. This sets `CMAKE_BUILD_TYPE` to `Debug`, `SUNDIALS_INDEX_SIZE` to `32` and `SUNDIALS_PRECISION` to `double`.
 Default: `OFF`

A.1.3 Configuration examples

The following examples will help demonstrate usage of the CMake configure options.

To configure SUNDIALS using the default C and Fortran compilers, and default `mpicc` and `mpif77` parallel compilers, enable compilation of examples, and install libraries, headers, and example sources under subdirectories of `/home/myname/sundials/`, use:

```
% cmake \  
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \  
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \  
> -DENABLE_MPI=ON \  
> -DFCMIX_ENABLE=ON \  
> /home/myname/sundials/solverdir  
%  
% make install  
%
```

To disable installation of the examples, use:

```
% cmake \  
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \  
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \  
> -DENABLE_MPI=ON \  
> -DFCMIX_ENABLE=ON \  
> -DEXAMPLES_INSTALL=OFF \  
> /home/myname/sundials/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 any external libraries used with SUNDIALS must also be build as a shared library or as a static library compiled with the `-fPIC` flag.



Building with LAPACK

To enable LAPACK, set the `ENABLE_LAPACK` 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 required for LAPACK.

```
% cmake \  
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \  
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \  
> -DENABLE_LAPACK=ON \  
> -DLAPACK_LIBRARIES=/mylapackpath/lib/libblas.so;/mylapackpath/lib/liblapack.so \  
> /home/myname/sundials/solverdir  
%  
% make install  
%
```

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>. SUNDIALS has been tested with SuiteSparse version 5.7.2. To enable KLU, set `ENABLE_KLU` to `ON`, set `KLU_INCLUDE_DIR` to the `include` path of the KLU installation and set `KLU_LIBRARY_DIR` to the `lib` path of the KLU installation. The CMake configure will result in populating the following variables: `AMD_LIBRARY`, `AMD_LIBRARY_DIR`, `BTF_LIBRARY`, `BTF_LIBRARY_DIR`, `COLAMD_LIBRARY`, `COLAMD_LIBRARY_DIR`, and `KLU_LIBRARY`.

Building with SuperLU_MT

The SuperLU_MT libraries are available for download from the Lawrence Berkeley National Laboratory website: http://crd-legacy.lbl.gov/~xiaoye/SuperLU/#superlu_mt. SUNDIALS has been tested with SuperLU_MT version 3.1. To enable SuperLU_MT, set `ENABLE_SUPERLUMT` 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_LIBRARIES` must be set to a semi-colon separated list of other libraries SuperLU_MT depends on. For example, if SuperLU_MT was build with an external blas library, then include the full path to the blas library in this list. Additionally, 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 `ENABLE_OPENMP` or `ENABLE_PTHREAD` set to `ON` then SuperLU_MT should be set to use the same threading type.



Building with SuperLU_DIST

The SuperLU_DIST libraries are available for download from the Lawrence Berkeley National Laboratory website: http://crd-legacy.lbl.gov/~xiaoye/SuperLU/#superlu_dist. SUNDIALS has been tested with SuperLU_DIST 6.1.1. To enable SuperLU_DIST, set `ENABLE_SUPERLUDIST` to `ON`, set `SUPERLUDIST_INCLUDE_DIR` to the `include` directory of the SuperLU_DIST installation (typically `SRC`), and set the variable

`SUPERLUDIST_LIBRARY_DIR` to the path to library directory of the SuperLU_DIST installation (typically `lib`). At the same time, the variable `SUPERLUDIST_LIBRARIES` must be set to a semi-colon separated list of other libraries SuperLU_DIST depends on. For example, if SuperLU_DIST was built with LAPACK, then include the LAPACK library in this list. If SuperLU_DIST was built with OpenMP support, then you may set `SUPERLUDIST_OPENMP` to `ON` to utilize the OpenMP functionality of SuperLU_DIST.

Do not mix thread types when building SUNDIALS solvers. If threading is enabled for SUNDIALS by having `ENABLE_PTHREAD` set to `ON` then SuperLU_DIST should not be set to use OpenMP.



Building with PETSc

The PETSc libraries are available for download from the Argonne National Laboratory website: <http://www.mcs.anl.gov/>. SUNDIALS has been tested with PETSc version 3.10.0–3.14.0. To enable PETSc, set `ENABLE_PETSC` to `ON` and then set `PETSC_DIR` to the path of the PETSc installation. Alternatively, a user can provide a list of include paths in `PETSC_INCLUDES`, and a list of complete paths to the libraries needed in `PETSC_LIBRARIES`.

Building with hypre

The *hypre* libraries are available for download from the Lawrence Livermore National Laboratory website: <http://computing.llnl.gov/projects/hypre>. SUNDIALS has been tested with *hypre* version 2.14.0–2.19.0. To enable *hypre*, set `ENABLE_HYPRE` 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.

Note: SUNDIALS must be configured so that `SUNDIALS_INDEX_SIZE` (or equivalently, `XSDK_INDEX_SIZE`) equals the precision of `HYPRE_BigInt` in the corresponding *hypre* installation.

Building with CUDA

SUNDIALS CUDA modules and examples have been tested with versions 9 through 11.0.2 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 `ENABLE_CUDA` 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 up to version 0.14.0. Building SUNDIALS RAJA modules requires a CUDA, HIP, or SYCL enabled RAJA installation. To enable RAJA, set `ENABLE_RAJA` to `ON`, set `SUNDIALS_RAJA_BACKENDS` to the desired backend (CUDA, HIP, or SYCL), and set `ENABLE_CUDA`, `ENABLE_HIP`, or `ENABLE_SYCL`, to `ON` depending on the selected backend. 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_CXX` to `ON`.

Building with Trilinos

Trilinos is a suite of numerical libraries developed by Sandia National Laboratories. It can be obtained at <https://github.com/trilinos/Trilinos>. SUNDIALS Trilinos modules and examples have been tested with Trilinos version 12.14.1 – 12.18.1. To enable Trilinos, set `ENABLE_TRILINOS` to `ON`. If Trilinos is installed in a nonstandard location you will be prompted to set the variable `Trilinos_DIR` with the path to the Trilinos CMake configuration file. It is desirable to build the Trilinos vector interface with same compiler and options that were used to build Trilinos. CMake will try to find the

correct compiler settings automatically from the Trilinos configuration file. If that is not successful, the compilers and options can be manually set with the following CMake variables:

- `Trilinos_INTERFACE_C_COMPILER`
- `Trilinos_INTERFACE_C_COMPILER_FLAGS`
- `Trilinos_INTERFACE_CXX_COMPILER`
- `Trilinos_INTERFACE_CXX_COMPILER_FLAGS`

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*/`CMAKE_INSTALL_LIBDIR` and *instdir*/`include`, respectively. The location can be changed by setting the CMake variable `CMAKE_INSTALL_PREFIX`. Although all installed libraries reside under *libdir*/`CMAKE_INSTALL_LIBDIR`, 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.

A.4.1 Using SUNDIALS as a Third Party Library in other CMake Projects

The `make install` command will also install a **CMake package configuration file** that other CMake projects can load to get all the information needed to build against SUNDIALS. In the consuming project's CMake code, the `find_package` command may be used to search for the configuration file, which will be installed to `instdir/SUNDIALS_INSTALL_CMAKEDIR/SUNDIALSConfig.cmake` alongside a package version file `instdir/SUNDIALS_INSTALL_CMAKEDIR/SUNDIALSConfigVersion.cmake`. Together these files contain all the information the consuming project needs to use SUNDIALS, including exported CMake targets. The SUNDIALS exported CMake targets follow the same naming convention as the generated library binaries, e.g. the exported target for CVODE is `SUNDIALS::cvode`. The CMake code snipped below shows how a consuming project might leverage the SUNDIALS package configuration file to build against SUNDIALS in their own CMake project.

```
project(MyProject)

# Set the variable SUNDIALS_DIR to the SUNDIALS instdir.
# When using the cmake CLI command, this can be done like so:
#   cmake -D SUNDIALS_DIR=/path/to/sundials/installation

find_package(SUNDIALS REQUIRED)
```

```
add_executable(myexec main.c)

# Link to SUNDIALS libraries through the exported targets.
# This is just an example, users should link to the targets appropriate
# for their use case.
target_link_libraries(myexec PUBLIC SUNDIALS::cvtode SUNDIALS::nvecpetsc)
```

Table A.1: SUNDIALS libraries and header files

SHARED	Libraries	n/a
	Header files	sundials/sundials_config.h sundials/sundials_fconfig.h sundials/sundials_types.h sundials/sundials_math.h sundials/sundials_nvector.h sundials/sundials_fnvector.h sundials/sundials_matrix.h sundials/sundials_linearsolver.h sundials/sundials_iterative.h sundials/sundials_direct.h sundials/sundials_dense.h sundials/sundials_band.h sundials/sundials_nonlinearsolver.h sundials/sundials_version.h sundials/sundials_mpi_types.h sundials/sundials_cuda_policies.hpp
NVECTOR_SERIAL	Libraries	libsundials_nvecserial.lib libsundials_fnvecserial_mod.lib libsundials_fnvecserial.a
	Header files	nvector/nvector_serial.h
	Module files	fnvector_serial_mod.mod
NVECTOR_PARALLEL	Libraries	libsundials_nvecparallel.lib libsundials_fnvecparallel.a libsundials_fnvecparallel_mod.lib
	Header files	nvector/nvector_parallel.h
	Module files	fnvector_parallel_mod.mod
NVECTOR_MANYVECTOR	Libraries	libsundials_nvecmanyvector.lib libsundials_nvecmanyvector_mod.lib
	Header files	nvector/nvector_manyvector.h
	Module files	fnvector_manyvector_mod.mod
NVECTOR_MPIMANYVECTOR	Libraries	libsundials_nvecmpimanyvector.lib libsundials_nvecmpimanyvector_mod.lib
	Header files	nvector/nvector_mpimanyvector.h
	Module files	fnvector_mpimanyvector_mod.mod
<i>continued on next page</i>		

<i>continued from last page</i>		
NVECTOR_MPIPLUSX	Libraries	libsundials_nvecmpiplusx. <i>lib</i> libsundials_nvecmpiplusx_mod. <i>lib</i>
	Header files	nvector/nvector_mpiplusx.h
	Module files	fnvector_mpiplusx_mod.mod
NVECTOR_OPENMP	Libraries	libsundials_nvecopenmp. <i>lib</i> libsundials_fnvecopenmp_mod. <i>lib</i> libsundials_fnvecopenmp.a
	Header files	nvector/nvector_openmp.h
	Module files	fnvector_openmp_mod.mod
NVECTOR_OPENMPDEV	Libraries	libsundials_nvecopenmpdev. <i>lib</i>
	Header files	nvector/nvector_openmpdev.h
NVECTOR_PTHREADS	Libraries	libsundials_nvecpthreads. <i>lib</i> libsundials_fnvecpthreads_mod. <i>lib</i> libsundials_fnvecpthreads.a
	Header files	nvector/nvector_pthreads.h
	Module files	fnvector_pthreads_mod.mod
NVECTOR_PARHYP	Libraries	libsundials_nvecparhyp. <i>lib</i>
	Header files	nvector/nvector_parhyp.h
NVECTOR_PETSC	Libraries	libsundials_nvecpetsc. <i>lib</i>
	Header files	nvector/nvector_petsc.h
NVECTOR_CUDA	Libraries	libsundials_nveccuda. <i>lib</i>
	Header files	nvector/nvector_cuda.h
NVECTOR_HIP	Libraries	libsundials_nvechhip. <i>lib</i>
	Header files	nvector/nvector_hip.h
NVECTOR_RAJA	Libraries	libsundials_nveccudaraja. <i>lib</i> libsundials_nvechpraja. <i>lib</i>
	Header files	nvector/nvector_rajah.h
NVECTOR_SYCL	Libraries	libsundials_nvecsycl. <i>lib</i>
	Header files	nvector/nvector_sycl.h
NVECTOR_TRILINOS	Libraries	libsundials_nvec trilinos. <i>lib</i>
	Header files	nvector/nvector_trilinos.h nvector/trilinos/SundialsTpetraVectorInterface.hpp nvector/trilinos/SundialsTpetraVectorKernels.hpp
SUNMATRIX_BAND	Libraries	libsundials_sunmatrixband. <i>lib</i> libsundials_fsunmatrixband_mod. <i>lib</i> libsundials_fsunmatrixband.a
	Header files	sunmatrix/sunmatrix_band.h
	Module files	fsunmatrix_band_mod.mod
<i>continued on next page</i>		

<i>continued from last page</i>		
SUNMATRIX_DENSE	Libraries	libsundials_sunmatrixdense. <i>lib</i> libsundials_fsunmatrixdense_mod. <i>lib</i> libsundials_fsunmatrixdense.a
	Header files	sunmatrix/sunmatrix_dense.h
	Module files	fsunmatrix_dense_mod.mod
SUNMATRIX_SPARSE	Libraries	libsundials_sunmatrixsparse. <i>lib</i> libsundials_fsunmatrixsparse_mod. <i>lib</i> libsundials_fsunmatrixsparse.a
	Header files	sunmatrix/sunmatrix_sparse.h
	Module files	fsunmatrix_sparse_mod.mod
SUNMATRIX_SLUNRLOC	Libraries	libsundials_sunmatrixslunloc. <i>lib</i>
	Header files	sunmatrix/sunmatrix_slunloc.h
SUNLINSOL_CUSPARSE	Libraries	libsundials_sunmatrixcusparse. <i>lib</i>
	Header files	sunmatrix/sunmatrix_cusparse.h
SUNLINSOL_BAND	Libraries	libsundials_sunlinsolband. <i>lib</i> libsundials_fsunlinsolband_mod. <i>lib</i> libsundials_fsunlinsolband.a
	Header files	sunlinsol/sunlinsol_band.h
	Module files	fsunlinsol_band_mod.mod
SUNLINSOL_DENSE	Libraries	libsundials_sunlinsoldense. <i>lib</i> libsundials_fsunlinsoldense_mod. <i>lib</i> libsundials_fsunlinsoldense.a
	Header files	sunlinsol/sunlinsol_dense.h
	Module files	fsunlinsol_dense_mod.mod
SUNLINSOL_KLU	Libraries	libsundials_sunlinsolklu. <i>lib</i> libsundials_fsunlinsolklu_mod. <i>lib</i> libsundials_fsunlinsolklu.a
	Header files	sunlinsol/sunlinsol_klu.h
	Module files	fsunlinsol_klu_mod.mod
SUNLINSOL_LAPACKBAND	Libraries	libsundials_sunlinsollapackband. <i>lib</i> libsundials_fsunlinsollapackband.a
	Header files	sunlinsol/sunlinsol_lapackband.h
SUNLINSOL_LAPACKDENSE	Libraries	libsundials_sunlinsollapackdense. <i>lib</i> libsundials_fsunlinsollapackdense.a
	Header files	sunlinsol/sunlinsol_lapackdense.h
SUNLINSOL_PCG	Libraries	libsundials_sunlinsolpcg. <i>lib</i> libsundials_fsunlinsolpcg_mod. <i>lib</i>
	<i>continued on next page</i>	

<i>continued from last page</i>		
		libsundials_fsuninsolpcg.a
	Header files	sunlinsol/sunlinsol_pcg.h
	Module files	fsunlinsol_pcg_mod.mod
SUNLINSOL_SPBCGS	Libraries	libsundials_sunlinsolspbcgs.lib libsundials_fsunlinsolspbcgs_mod.lib libsundials_fsunlinsolspbcgs.a
	Header files	sunlinsol/sunlinsol_spbcgs.h
	Module files	fsunlinsol_spbcgs_mod.mod
SUNLINSOL_SPGMR	Libraries	libsundials_sunlinsolspgmr.lib libsundials_fsunlinsolspgmr_mod.lib libsundials_fsunlinsolspgmr.a
	Header files	sunlinsol/sunlinsol_spgmr.h
	Module files	fsunlinsol_spgmr_mod.mod
SUNLINSOL_SPGMR	Libraries	libsundials_sunlinsolspgmr.lib libsundials_fsunlinsolspgmr_mod.lib libsundials_fsunlinsolspgmr.a
	Header files	sunlinsol/sunlinsol_spgmr.h
	Module files	fsunlinsol_spgmr_mod.mod
SUNLINSOL_SPTFQMR	Libraries	libsundials_sunlinsolsptfqmr.lib libsundials_fsunlinsolsptfqmr_mod.lib libsundials_fsunlinsolsptfqmr.a
	Header files	sunlinsol/sunlinsol_sptfqmr.h
	Module files	fsunlinsol_sptfqmr_mod.mod
SUNLINSOL_SUPERLUMT	Libraries	libsundials_sunlinsolsuperlumt.lib libsundials_fsunlinsolsuperlumt.a
	Header files	sunlinsol/sunlinsol_superlumt.h
SUNLINSOL_SUPERLUDIST	Libraries	libsundials_sunlinsolsuperludist.lib
	Header files	sunlinsol/sunlinsol_superludist.h
SUNLINSOL_CUSOLVERSP_BATCHQR	Libraries	libsundials_sunlinsolcusolversp.lib
	Header files	sunlinsol/sunlinsol_cusolverp_batchqr.h
SUNNONLINSOL_NEWTON	Libraries	libsundials_sunnonlinsolnewton.lib libsundials_fsunnonlinsolnewton_mod.lib libsundials_fsunnonlinsolnewton.a
	Header files	sunnonlinsol/sunnonlinsol_newton.h
	Module files	fsunnonlinsol_newton_mod.mod
SUNNONLINSOL_FIXEDPOINT	Libraries	libsundials_sunnonlinsolfixedpoint.lib

continued on next page

<i>continued from last page</i>			
		libsundials_fsunnnonlinsolfixedpoint.a libsundials_fsunnnonlinsolfixedpoint_mod. <i>lib</i>	
	Header files	sunnnonlinsol/sunnnonlinsol_fixedpoint.h	
	Module files	fsunnnonlinsol_fixedpoint_mod.mod	
SUNNONLINSOL_PETSCSNES	Libraries	libsundials_sunnnonlinsolpetscsnes. <i>lib</i>	
	Header files	sunnnonlinsol/sunnnonlinsol_petscsnes.h	
CVODE	Libraries	libsundials_cvode. <i>lib</i> libsundials_fcvcde.a libsundials_fcvcde_mod. <i>lib</i>	
	Header files	cvode/cvode.h cvode/cvode_direct.h cvode/cvode_spils.h cvode/cvode_bbdpre.h	cvode/cvode_impl.h cvode/cvode_ls.h cvode/cvode_bandpre.h
	Module files	fcvcde_mod.mod	
CVODES	Libraries	libsundials_cvodes. <i>lib</i> libsundials_fcvcodes_mod. <i>lib</i>	
	Header files	cvodes/cvodes.h cvodes/cvodes_direct.h cvodes/cvodes_spils.h cvodes/cvodes_bbdpre.h	cvodes/cvodes_impl.h cvodes/cvodes_ls.h cvodes/cvodes_bandpre.h
	Module files	fcvcodes_mod.mod	
ARKODE	Libraries	libsundials_arkode. <i>lib</i> libsundials_farkode.a libsundials_farkode_mod. <i>lib</i>	
	Header files	arkode/arkode.h arkode/arkode_ls.h arkode/arkode_bbdpre.h	arkode/arkode_impl.h arkode/arkode_bandpre.h
	Module files	farkode_mod.mod farkode_erkstep_mod.mod	farkode_arkstep_mod.mod farkode_mrstep_mod.mod
IDA	Libraries	libsundials_ida. <i>lib</i> libsundials_fida.a libsundials_fida_mod. <i>lib</i>	
	Header files	ida/ida.h ida/ida_direct.h ida/ida_spils.h	ida/ida_impl.h ida/ida_ls.h ida/ida_bbdpre.h
	Module files	fida_mod.mod	
			<i>continued on next page</i>

<i>continued from last page</i>		
IDAS	Libraries	libsundials_idas. <i>lib</i> libsundials_fdas_mod. <i>lib</i>
	Header files	idas/idas.h idas/idas_impl.h idas/idas_direct.h idas/idas_ls.h idas/idas_spils.h idas/idas_bbdpre.h
	Module files	fidas_mod.mod
KINSOL	Libraries	libsundials_kinsol. <i>lib</i> libsundials_fkinsol.a libsundials_fkinsol_mod. <i>lib</i>
	Header files	kinsol/kinsol.h kinsol/kinsol_impl.h kinsol/kinsol_direct.h kinsol/kinsol_ls.h kinsol/kinsol_spils.h kinsol/kinsol_bbdpre.h
	Module files	fkinsol_mod.mod

Appendix B

KINSOL Constants

Below we list all input and output constants used by the main solver and linear solver modules, together with their numerical values and a short description of their meaning.

B.1 KINSOL input constants

KINSOL main solver module		
KIN_ETACHOICE1	1	Use Eisenstat and Walker Choice 1 for η .
KIN_ETACHOICE2	2	Use Eisenstat and Walker Choice 2 for η .
KIN_ETACONSTANT	3	Use constant value for η .
KIN_NONE	0	Use inexact Newton globalization.
KIN_LINESEARCH	1	Use linesearch globalization.

Iterative linear solver modules		
PREC_NONE	0	No preconditioning
PREC_RIGHT	2	Preconditioning on the right.
MODIFIED_GS	1	Use modified Gram-Schmidt procedure.
CLASSICAL_GS	2	Use classical Gram-Schmidt procedure.

B.2 KINSOL output constants

KINSOL main solver module		
KIN_SUCCESS	0	Successful function return.
KIN_INITIAL_GUESS_OK	1	The initial user-supplied guess already satisfies the stopping criterion.
KIN_STEP_LT_STPTOL	2	The stopping tolerance on scaled step length was satisfied.
KIN_WARNING	99	A non-fatal warning. The solver will continue.
KIN_MEM_NULL	-1	The <code>kin_mem</code> argument was NULL.
KIN_ILL_INPUT	-2	One of the function inputs is illegal.
KIN_NO_MALLOC	-3	The KINSOL memory was not allocated by a call to <code>KINMalloc</code> .
KIN_MEM_FAIL	-4	A memory allocation failed.

KIN_LINESEARCH_NONCONV	-5	The linesearch algorithm was unable to find an iterate sufficiently distinct from the current iterate.
KIN_MAXITER_REACHED	-6	The maximum number of nonlinear iterations has been reached.
KIN_MXNEWT_5X_EXCEEDED	-7	Five consecutive steps have been taken that satisfy a scaled step length test.
KIN_LINESEARCH_BCFAIL	-8	The linesearch algorithm was unable to satisfy the β -condition for <code>nbcbfails</code> iterations.
KIN_LINSOLV_NO_RECOVERY	-9	The user-supplied routine preconditioner slve function failed recoverably, but the preconditioner is already current.
KIN_LINIT_FAIL	-10	The linear solver's initialization function failed.
KIN_LSETUP_FAIL	-11	The linear solver's setup function failed in an unrecoverable manner.
KIN_LSOLVE_FAIL	-12	The linear solver's solve function failed in an unrecoverable manner.
KIN_SYSFUNC_FAIL	-13	The system function failed in an unrecoverable manner.
KIN_FIRST_SYSFUNC_ERR	-14	The system function failed recoverably at the first call.
KIN_REPTD_SYSFUNC_ERR	-15	The system function had repeated recoverable errors.

KINLS **linear solver interface**

KINLS_SUCCESS	0	Successful function return.
KINLS_MEM_NULL	-1	The <code>kin_mem</code> argument was NULL.
KINLS_LMEM_NULL	-2	The KINLS linear solver has not been initialized.
KINLS_ILL_INPUT	-3	The KINLS solver is not compatible with the current NVECTOR module, or an input value was illegal.
KINLS_MEM_FAIL	-4	A memory allocation request failed.
KINLS_PMEM_NULL	-5	The preconditioner module has not been initialized.
KINLS_JACFUNC_ERR	-6	The Jacobian function failed
KINLS_SUNMAT_FAIL	-7	An error occurred with the current SUNMATRIX module.
KINLS_SUNLS_FAIL	-8	An error occurred with the current SUNLINSOL module.

Appendix C

SUNDIALS Release History

Table C.1: Release History

Date	SUNDIALS	ARKODE	CVODE	CVODES	IDA	IDAS	KINSOL
Sep 2021	5.8.0	4.8.0	5.8.0	5.8.0	5.8.0	4.8.0	5.8.0
Jan 2021	5.7.0	4.7.0	5.7.0	5.7.0	5.7.0	4.7.0	5.7.0
Dec 2020	5.6.1	4.6.1	5.6.1	5.6.1	5.6.1	4.6.1	5.6.1
Dec 2020	5.6.0	4.6.0	5.6.0	5.6.0	5.6.0	4.6.0	5.6.0
Oct 2020	5.5.0	4.5.0	5.5.0	5.5.0	5.5.0	4.5.0	5.5.0
Sep 2020	5.4.0	4.4.0	5.4.0	5.4.0	5.4.0	4.4.0	5.4.0
May 2020	5.3.0	4.3.0	5.3.0	5.3.0	5.3.0	4.3.0	5.3.0
Mar 2020	5.2.0	4.2.0	5.2.0	5.2.0	5.2.0	4.2.0	5.2.0
Jan 2020	5.1.0	4.1.0	5.1.0	5.1.0	5.1.0	4.1.0	5.1.0
Oct 2019	5.0.0	4.0.0	5.0.0	5.0.0	5.0.0	4.0.0	5.0.0
Feb 2019	4.1.0	3.1.0	4.1.0	4.1.0	4.1.0	3.1.0	4.1.0
Jan 2019	4.0.2	3.0.2	4.0.2	4.0.2	4.0.2	3.0.2	4.0.2
Dec 2018	4.0.1	3.0.1	4.0.1	4.0.1	4.0.1	3.0.1	4.0.1
Dec 2018	4.0.0	3.0.0	4.0.0	4.0.0	4.0.0	3.0.0	4.0.0
Oct 2018	3.2.1	2.2.1	3.2.1	3.2.1	3.2.1	2.2.1	3.2.1
Sep 2018	3.2.0	2.2.0	3.2.0	3.2.0	3.2.0	2.2.0	3.2.0
Jul 2018	3.1.2	2.1.2	3.1.2	3.1.2	3.1.2	2.1.2	3.1.2
May 2018	3.1.1	2.1.1	3.1.1	3.1.1	3.1.1	2.1.1	3.1.1
Nov 2017	3.1.0	2.1.0	3.1.0	3.1.0	3.1.0	2.1.0	3.1.0
Sep 2017	3.0.0	2.0.0	3.0.0	3.0.0	3.0.0	2.0.0	3.0.0
Sep 2016	2.7.0	1.1.0	2.9.0	2.9.0	2.9.0	1.3.0	2.9.0
Aug 2015	2.6.2	1.0.2	2.8.2	2.8.2	2.8.2	1.2.2	2.8.2
Mar 2015	2.6.1	1.0.1	2.8.1	2.8.1	2.8.1	1.2.1	2.8.1

continued on next page

<i>continued from last page</i>								
Date	SUNDIALS	ARKODE	CVODE	CVODES	IDA	IDAS	KINSOL	
Mar 2015	2.6.0	1.0.0	2.8.0	2.8.0	2.8.0	1.2.0	2.8.0	
Mar 2012	2.5.0	–	2.7.0	2.7.0	2.7.0	1.1.0	2.7.0	
May 2009	2.4.0	–	2.6.0	2.6.0	2.6.0	1.0.0	2.6.0	
Nov 2006	2.3.0	–	2.5.0	2.5.0	2.5.0	–	2.5.0	
Mar 2006	2.2.0	–	2.4.0	2.4.0	2.4.0	–	2.4.0	
May 2005	2.1.1	–	2.3.0	2.3.0	2.3.0	–	2.3.0	
Apr 2005	2.1.0	–	2.3.0	2.2.0	2.3.0	–	2.3.0	
Mar 2005	2.0.2	–	2.2.2	2.1.2	2.2.2	–	2.2.2	
Jan 2005	2.0.1	–	2.2.1	2.1.1	2.2.1	–	2.2.1	
Dec 2004	2.0.0	–	2.2.0	2.1.0	2.2.0	–	2.2.0	
Jul 2002	1.0.0	–	2.0.0	1.0.0	2.0.0	–	2.0.0	
Mar 2002	–	–	1.0.0 ³	–	–	–	–	
Feb 1999	–	–	–	–	1.0.0 ⁴	–	–	
Aug 1998	–	–	–	–	–	–	1.0.0 ⁵	
Jul 1997	–	–	1.0.0 ²	–	–	–	–	
Sep 1994	–	–	1.0.0 ¹	–	–	–	–	

¹CVODE written, ²PVODE written, ³CVODE and PVODE combined, ⁴IDA written, ⁵KINSOL written

Bibliography

- [1] AMD ROCm Documentation. <https://rocmdocs.amd.com/en/latest/index.html>.
- [2] Intel oneAPI Programming Guide. <https://software.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html>.
- [3] KLU Sparse Matrix Factorization Library. <http://faculty.cse.tamu.edu/davis/suitesparse.html>.
- [4] MAGMA: Matrix Algebra on GPU and Multicore Architectures. <https://icl.utk.edu/magma/index.html>.
- [5] NVIDIA CUDA Programming Guide. <https://docs.nvidia.com/cuda/index.html>.
- [6] NVIDIA cuSOLVER Programming Guide. <https://docs.nvidia.com/cuda/cusolver/index.html>.
- [7] NVIDIA cuSPARSE Programming Guide. <https://docs.nvidia.com/cuda/cusparse/index.html>.
- [8] SuperLU_DIST Parallel Sparse Matrix Factorization Library. <http://crd-legacy.lbl.gov/~xiaoye/-SuperLU/>.
- [9] SuperLU_MT Threaded Sparse Matrix Factorization Library. <http://crd-legacy.lbl.gov/~xiaoye/-SuperLU/>.
- [10] D. G. Anderson. Iterative procedures for nonlinear integral equations. *J. Assoc. Comput. Machinery*, 12:547–560, 1965.
- [11] P. N. Brown. A local convergence theory for combined inexact-Newton/finite difference projection methods. *SIAM J. Numer. Anal.*, 24(2):407–434, 1987.
- [12] P. N. Brown and A. C. Hindmarsh. Reduced Storage Matrix Methods in Stiff ODE Systems. *J. Appl. Math. & Comp.*, 31:49–91, 1989.
- [13] P. N. Brown and Y. Saad. Hybrid Krylov Methods for Nonlinear Systems of Equations. *SIAM J. Sci. Stat. Comput.*, 11:450–481, 1990.
- [14] G. D. Byrne. Pragmatic Experiments with Krylov Methods in the Stiff ODE Setting. In J.R. Cash and I. Gladwell, editors, *Computational Ordinary Differential Equations*, pages 323–356, Oxford, 1992. Oxford University Press.
- [15] A. M. Collier and R. Serban. Example Programs for KINSOL v5.8.0. Technical Report UCRL-SM-208114, LLNL, 2021.
- [16] T. A. Davis and P. N. Ekanathan. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Softw.*, 37(3), 2010.
- [17] R. S. Dembo, S. C. Eisenstat, and T. Steihaug. Inexact Newton Methods. *SIAM J. Numer. Anal.*, 19:400–408, 1982.
- [18] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.

- [19] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM, Philadelphia, 1996.
- [20] M.R. Dorr, J.-L. Fattebert, M.E. Wickett, J.F. Belak, and P.E.A. Turchi. A numerical algorithm for the solution of a phase-field model of polycrystalline materials. *Journal of Computational Physics*, 229(3):626–641, 2010.
- [21] S. C. Eisenstat and H. F. Walker. Choosing the Forcing Terms in an Inexact Newton Method. *SIAM J. Sci. Comput.*, 17:16–32, 1996.
- [22] H. Fang and Y. Saad. Two classes of secant methods for nonlinear acceleration. *Numer. Linear Algebra Appl.*, 16:197–221, 2009.
- [23] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14:470–482, 1993.
- [24] Laura Grigori, James W. Demmel, and Xiaoye S. Li. Parallel symbolic factorization for sparse LU with static pivoting. *SIAM J. Scientific Computing*, 29(3):1289–1314, 2007.
- [25] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *J. Research of the National Bureau of Standards*, 49(6):409–436, 1952.
- [26] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS, suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, (31):363–396, 2005.
- [27] A. C. Hindmarsh, R. Serban, and A. Collier. Example Programs for IDA v5.8.0. Technical Report UCRL-SM-208113, LLNL, 2021.
- [28] A. C. Hindmarsh, R. Serban, and D. R. Reynolds. Example Programs for CVODE v5.8.0. Technical report, LLNL, 2021. UCRL-SM-208110.
- [29] Seth R. Johnson, Andrey Prokopenko, and Katherine J. Evans. Automated fortran-c++ bindings for large-scale scientific applications. arXiv:1904.02546 [cs], 2019.
- [30] C. T. Kelley. *Iterative Methods for Solving Linear and Nonlinear Equations*. SIAM, Philadelphia, 1995.
- [31] X. S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.
- [32] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.
- [33] X.S. Li, J.W. Demmel, J.R. Gilbert, L. Grigori, M. Shao, and I. Yamazaki. SuperLU Users’ Guide. Technical Report LBNL-44289, Lawrence Berkeley National Laboratory, September 1999. <http://crd.lbl.gov/~xiaoye/SuperLU/>. Last update: August 2011.
- [34] P. A. Lott, H. F. Walker, C. S. Woodward, and U. M. Yang. An accelerated Picard method for nonlinear systems related to variably saturated flow. *Adv. Wat. Resour.*, 38:92–101, 2012.
- [35] J. M. Ortega and W. C. Rheinbolt. *Iterative solution of nonlinear equations in several variables*. SIAM, Philadelphia, 2000. Originally published in 1970 by Academic Press.
- [36] Daniel R. Reynolds. Example Programs for ARKODE v4.8.0. Technical report, Southern Methodist University, 2021.
- [37] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14(2):461–469, 1993.

-
- [38] Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 7:856–869, 1986.
- [39] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010.
- [40] H. A. Van Der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 13:631–644, 1992.
- [41] H. F. Walker and P. Ni. Anderson acceleration for fixed-point iterations. *SIAM Jour. Num. Anal.*, 49(4):1715–1735, 2011.

Index

- Anderson acceleration
 - definition, [24](#)
- Anderson acceleration UA
 - definition, [24](#)
- ARKStepGetCurrentGamma, [240](#)
- ARKStepGetNonlinearSystemData, [240](#)

- BIG_REAL, [32](#), [98](#), [105](#)
- booleantype, [32](#)

- CVodeGetCurrentGamma, [240](#)
- CVodeGetNonlinearSystemData, [240](#)

- data types
 - Fortran, [77](#)

- eh_data, [62](#)
- error message
 - user-defined handler, [42](#)
- error messages, [40](#)
 - redirecting, [42](#)
 - user-defined handler, [62](#)
- ETA_CONST, [85](#)
- ETA_FORM, [85](#)
- ETA_PARAMS, [85](#)

- Fixed-point iteration
 - definition, [23](#)
- fixed-point system
 - definition, [19](#)
- FKBJAC, [82](#)
- FKCOMMFN, [88](#)
- FKDJAC, [82](#)
- FKFUN, [79](#)
- FKINBANDSETJAC, [82](#)
- FKINBBD interface module
 - interface to the KINBBDPRE module, [85](#)
- FKINBBDINIT, [87](#)
- FKINBBDOPT, [88](#)
- FKINCREATE, [81](#)
- FKINDENSESETJAC, [82](#)
- FKINDLSINIT, [81](#)
- FKINFREE, [84](#)
- FKININIT, [81](#)
- FKINJTIMES, [83](#), [88](#)
- FKINLSINIT, [81](#)
- FKINLSSETJAC, [83](#), [87](#)
- FKINLSSETPREC, [84](#)
- FKINSETIIN, [85](#)
- FKINSETRIN, [85](#)
- FKINSETVIN, [85](#)
- FKINSOL, [84](#)
- FKINSOL interface module
 - interface to the KINBBDPRE module, [88](#)
 - optional input and output, [85](#)
 - usage, [79–84](#)
 - user-callable functions, [78–79](#)
 - user-supplied functions, [79](#)
- fkinsol_mod, [71](#)
- FKINSPARSESETJAC, [83](#)
- FKINSPILSETJAC, [83](#)
- FKINSPILSETPREC, [84](#)
- FKINSPILSINIT, [81](#)
- FKINSPJAC, [82](#)
- FKLOCFN, [88](#)
- FKPSET, [84](#)
- FKPSOL, [83](#)
- FNORM_TOL, [85](#)
- fnvector_serial_mod, [119](#)
- FSUNBANDLINSOLINIT, [247](#)
- FSUNDENSELINSOLINIT, [244](#)
- FSUNKLUINIT, [256](#)
- FSUNKLUREINIT, [257](#)
- FSUNKLUSETORDERING, [257](#)
- FSUNLAPACKBANDINIT, [251](#)
- FSUNLAPACKDENSEINIT, [249](#)
- fsunlinsol_band_mod, [246](#)
- fsunlinsol_dense_mod, [244](#)
- fsunlinsol_klu_mod, [256](#)
- fsunlinsol_pcg_mod, [303](#)
- fsunlinsol_spgmrs_mod, [290](#)
- fsunlinsol_spgmrm_mod, [283](#)
- fsunlinsol_spgmrm_mod, [275](#)
- fsunlinsol_sptfqmr_mod, [296](#)
- FSUNMASSBANDLINSOLINIT, [247](#)
- FSUNMASSDENSELINSOLINIT, [244](#)
- FSUNMASSKLUINIT, [256](#)

- FSUNMASSKLUREINIT, 257
- FSUNMASSKLUSETORDERING, 258
- FSUNMASSLAPACKBANDINIT, 252
- FSUNMASSLAPACKDENSEINIT, 249
- FSUNMASSPCGINIT, 304
- FSUNMASSPCGSETMAXL, 305
- FSUNMASSPCGSETPRECTYPE, 304
- FSUNMASSSPBCGSINIT, 291
- FSUNMASSSPBCGSSETMAXL, 292
- FSUNMASSSPBCGSSETPRECTYPE, 291
- FSUNMASSSPFGMRINIT, 283
- FSUNMASSSPFGMRSETGSTYPE, 284
- FSUNMASSSPFGMRSETMAXRS, 285
- FSUNMASSSPFGMRSETPRECTYPE, 285
- FSUNMASSSPGMRINIT, 276
- FSUNMASSSPGMRSETGSTYPE, 276
- FSUNMASSSPGMRSETMAXRS, 277
- FSUNMASSSPGMRSETPRECTYPE, 277
- FSUNMASSSPTFQMRINIT, 297
- FSUNMASSSPTFQMRSETMAXL, 298
- FSUNMASSSPTFQMRSETPRECTYPE, 298
- FSUNMASSSUPERLUMTINIT, 265
- FSUNMASSSUPERLUMTSETORDERING, 266
- fsunmatrix_band_mod, 203
- fsunmatrix_dense_mod, 197
- fsunmatrix_sparse_mod, 210
- FSUNPCGINIT, 304
- FSUNPCGSETMAXL, 305
- FSUNPCGSETPRECTYPE, 304
- FSUNSPBCGSINIT, 290
- FSUNSPBCGSSETMAXL, 292
- FSUNSPBCGSSETPRECTYPE, 291
- FSUNSPFGMRINIT, 283
- FSUNSPFGMRSETGSTYPE, 284
- FSUNSPFGMRSETMAXRS, 285
- FSUNSPFGMRSETPRECTYPE, 284
- FSUNSPGMRINIT, 276
- FSUNSPGMRSETGSTYPE, 276
- FSUNSPGMRSETMAXRS, 277
- FSUNSPGMRSETPRECTYPE, 277
- FSUNSPTFQMRINIT, 297
- FSUNSPTFQMRSETMAXL, 298
- FSUNSPTFQMRSETPRECTYPE, 297
- FSUNSUPERLUMTINIT, 265
- FSUNSUPERLUMTSETORDERING, 265
- half-bandwidths, 69
- header files, 33, 68
- IDAGetCurrentCj, 240
- IDAGetNonlinearSystemData, 240
- ih_data, 62
- Inexact Newton iteration
 - definition, 19
- info message
 - user-defined handler, 42
- info messages
 - redirecting, 42
- informational messages
 - user-defined handler, 62
- IOUT, 85, 86
- Jacobian approximation function
 - band
 - use in FKINSOL, 82
 - dense
 - use in FKINSOL, 82
 - difference quotient, 52
 - Jacobian times vector
 - alternative-sys, 53
 - difference quotient, 52
 - user-supplied, 52, 64–65
 - sparse
 - use in FKINSOL, 82
 - user-supplied, 52, 62–64
- KIN_ETACHOICE1, 45
- KIN_ETACHOICE2, 45
- KIN_ETACONSTANT, 45
- KIN_FIRST_SYSFUNC_ERR, 40
- KIN_FP, 39
- KIN_ILL_INPUT, 37, 39, 43–52
- KIN_INITIAL_GUESS_OK, 39
- KIN_LINESEARCH, 39
- KIN_LINESEARCH_BCFAIL, 40
- KIN_LINESEARCH_NONCONV, 39
- KIN_LINIT_FAIL, 40
- KIN_LINSOLV_NO_RECOVERY, 40
- KIN_LSETUP_FAIL, 40
- KIN_LSOLVE_FAIL, 40
- KIN_MAXITER_REACHED, 40
- KIN_MEM_FAIL, 37, 39
- KIN_MEM_NULL, 37, 39, 42–52, 56, 57
- KIN_MXNEWT_5X_EXCEEDED, 40
- KIN_NO_MALLOC, 39
- KIN_NONE, 39
- KIN_PICARD, 39
- KIN_REPTD_SYSFUNC_ERR, 40
- KIN_STEP_LT_STPTOL, 39
- KIN_SUCCESS, 37, 39, 42–52, 56, 57
- KIN_SYSFUNC_FAIL, 40
- KIN_WARNING, 62
- KINBBDPRE preconditioner
 - optional output, 69–70
 - usage, 68–69
 - user-callable functions, 69
 - user-supplied functions, 67–68
- KINBBDPrecGetNumGfnEvals, 70

- KINBBDPrecGetWorkSpace, 70
- KINBBDPrecInit, 69
- KINCreate, 37
- KINDlsGetLastFlag, 61
- KINDlsGetNumFuncEvals, 58
- KINDlsGetNumJacEvals, 58
- KINDlsGetReturnFlagName, 61
- KINDlsGetWorkSpace, 58
- KINDlsJacFn, 64
- KINDlsSetJacFn, 52
- KINDlsSetLinearSolver, 38
- KINErrorHandlerFn, 62
- KINFree, 37
- KINGetFuncNorm, 57
- KINGetLastLinFlag, 60
- KINGetLinReturnFlagName, 61
- KINGetLinWorkSpace, 57
- KINGetNumBacktrackOps, 57
- KINGetNumBetaCondFails, 56
- KINGetNumFuncEvals, 56
- KINGetNumJacEvals, 58
- KINGetNumJtimesEvals, 60
- KINGetNumLinConvFails, 59
- KINGetNumLinFuncEvals, 58
- KINGetNumLinIters, 59
- KINGetNumNonlinSolvIters, 56
- KINGetNumPrecEvals, 59
- KINGetNumPrecSolves, 60
- KINGetStepLength, 57
- KINGetWorkSpace, 56
- KINInfoHandlerFn, 62
- KINInit, 37, 51
- KINLS linear solver
 - Jacobian-vector product approximation used by, 52
 - memory requirements, 57
- KINLS linear solver interface
 - Jacobian approximation used by, 52
 - optional input, 52–54
 - optional output, 57–61
 - preconditioner setup function, 54, 66
 - preconditioner solve function, 54, 65
 - use in FKINSOL, 81
- KINLS_ILL_INPUT, 38, 53, 69
- KINLS_LMEM_NULL, 52–54, 58–60, 69
- KINLS_MEM_FAIL, 38, 69
- KINLS_MEM_NULL, 38, 52–54, 58–60
- KINLS_PMEM_NULL, 70
- KINLS_SUCCESS, 38, 52–54, 58–60
- KINLS_SUNLS_FAIL, 38, 53, 54
- KINLsJacFn, 62
- KINLsJacTimesVecFn, 64
- KINLsPrecSetupFn, 66
- KINLsPrecSolveFn, 65
- KINSetConstraints, 49
- KINSetDamping, 50
- KINSetDampingAA, 51
- KINSetDelayAA, 51
- KINSetErrFile, 42
- KINSetErrHandlerFn, 42
- KINSetEtaConstValue, 46
- KINSetEtaForm, 45
- KINSetEtaParams, 46
- KINSetFuncNormTol, 48
- KINSetInfoFile, 42
- KINSetInfoHandlerFn, 43
- KINSetJacFn, 52
- KINSetJacTimesVecFn, 52
- KINSetJacTimesVecSysFn, 53
- KINSetLinearSolver, 35, 38, 62, 193
- KINSetMAA, 51
- KINSetMaxBetaFails, 48
- KINSetMaxNewtonStep, 47
- KINSetMaxSetupCalls, 45
- KINSetMaxSubSetupCalls, 45
- KINSetNoInitSetup, 44
- KINSetNoMinEps, 47
- KINSetNoResMon, 44
- KINSetNumMaxIters, 44
- KINSetPreconditioner, 53, 54
- KINSetPrintLevel, 43
- KINSetRelErrFunc, 48
- KINSetResMonConstValue, 47
- KINSetResMonParams, 47
- KINSetReturnNewest, 50
- KINSetScaledStepTol, 49
- KINSetSysFunc, 49
- KINSetUserData, 43
- KINSOL
 - brief description of, 1
 - motivation for writing in C, 2
 - package structure, 28
 - relationship to NKSOL, 1
- KINSOL linear solver interface
 - KINLS, 38
- KINSOL linear solver interfaces, 29
- KINSOL linear solvers
 - header files, 33
 - implementation details, 29
 - NVECTOR compatibility, 31
 - selecting one, 38
- KINSol, 35, 39
- kinsol/kinsol.h, 33
- kinsol/kinsol_ls.h, 33
- KINSOLKINSOL linear solvers
 - selecting one, 38
- KINSpilsGetLastFlag, 61
- KINSpilsGetNumConvFails, 59

- KINSpilsGetNumFuncEvals, 58
- KINSpilsGetNumJtimesEvals, 60
- KINSpilsGetNumLinIters, 59
- KINSpilsGetNumPrecEvals, 59
- KINSpilsGetNumPrecSolves, 60
- KINSpilsGetReturnFlagName, 61
- KINSpilsGetWorkspace, 58
- KINSpilsJacTimesVecFn, 65
- KINSpilsPrecSetupFn, 66
- KINSpilsPrecSolveFn, 65
- KINSpilsSetJacTimesVecFn, 53
- KINSpilsSetLinearSolver, 38
- KINSpilsSetPreconditioner, 54
- KINSysFn, 37, 53, 61

- MAA, 85
- MAX_NITERS, 85
- MAX_SETUPS, 85
- MAX_SP_SETUPS, 85
- MAX_STEP, 85
- memory requirements
 - KINBBDPRE preconditioner, 70
 - KINLS linear solver, 57
 - KINSOL solver, 56
- Modified Newton iteration
 - definition, 19
- MRIStepGetCurrentGamma, 240

- N_VCloneVectorArray, 106
- N_VCloneVectorArray_OpenMP, 126
- N_VCloneVectorArray_OpenMPDEV, 167
- N_VCloneVectorArray_Parallel, 121
- N_VCloneVectorArray_ParHyp, 136
- N_VCloneVectorArray_Petsc, 140
- N_VCloneVectorArray_Pthreads, 132
- N_VCloneVectorArray_Serial, 115
- N_VCloneVectorArrayEmpty, 106
- N_VCloneVectorArrayEmpty_OpenMP, 126
- N_VCloneVectorArrayEmpty_OpenMPDEV, 167
- N_VCloneVectorArrayEmpty_Parallel, 121
- N_VCloneVectorArrayEmpty_ParHyp, 136
- N_VCloneVectorArrayEmpty_Petsc, 140
- N_VCloneVectorArrayEmpty_Pthreads, 132
- N_VCloneVectorArrayEmpty_Serial, 116
- N_VCopyFromDevice_Cuda, 145
- N_VCopyFromDevice_Hip, 151
- N_VCopyFromDevice_OpenMPDEV, 168
- N_VCopyFromDevice_Raja, 157
- N_VCopyFromDevice_Sycl, 161
- N_VCopyOps, 106
- N_VCopyToDevice_Cuda, 145
- N_VCopyToDevice_Hip, 151
- N_VCopyToDevice_OpenMPDEV, 168
- N_VCopyToDevice_Raja, 157
- N_VCopyToDevice_Sycl, 161
- N_VDestroyVectorArray, 106
- N_VDestroyVectorArray_OpenMP, 127
- N_VDestroyVectorArray_OpenMPDEV, 167
- N_VDestroyVectorArray_Parallel, 121
- N_VDestroyVectorArray_ParHyp, 136
- N_VDestroyVectorArray_Petsc, 140
- N_VDestroyVectorArray_Pthreads, 132
- N_VDestroyVectorArray_Serial, 116
- N_Vector, 33, 91, 108
- N_VEnableConstVectorArray_Cuda, 146
- N_VEnableConstVectorArray_Hip, 152
- N_VEnableConstVectorArray_ManyVector, 175
- N_VEnableConstVectorArray_MPIManyVector, 180
- N_VEnableConstVectorArray_OpenMP, 128
- N_VEnableConstVectorArray_OpenMPDEV, 169
- N_VEnableConstVectorArray_Parallel, 123
- N_VEnableConstVectorArray_ParHyp, 138
- N_VEnableConstVectorArray_Petsc, 141
- N_VEnableConstVectorArray_Pthreads, 134
- N_VEnableConstVectorArray_Raja, 158
- N_VEnableConstVectorArray_Serial, 117
- N_VEnableConstVectorArray_Sycl, 163
- N_VEnableDotProdMulti_Cuda, 146
- N_VEnableDotProdMulti_Hip, 152
- N_VEnableDotProdMulti_ManyVector, 174
- N_VEnableDotProdMulti_MPIManyVector, 179
- N_VEnableDotProdMulti_OpenMP, 128
- N_VEnableDotProdMulti_OpenMPDEV, 169
- N_VEnableDotProdMulti_Parallel, 123
- N_VEnableDotProdMulti_ParHyp, 137
- N_VEnableDotProdMulti_Petsc, 141
- N_VEnableDotProdMulti_Pthreads, 133
- N_VEnableDotProdMulti_Serial, 117
- N_VEnableFusedOps_Cuda, 146
- N_VEnableFusedOps_Hip, 152
- N_VEnableFusedOps_ManyVector, 174
- N_VEnableFusedOps_MPIManyVector, 179
- N_VEnableFusedOps_OpenMP, 127
- N_VEnableFusedOps_OpenMPDEV, 168
- N_VEnableFusedOps_Parallel, 122
- N_VEnableFusedOps_ParHyp, 137
- N_VEnableFusedOps_Petsc, 140
- N_VEnableFusedOps_Pthreads, 132
- N_VEnableFusedOps_Raja, 158
- N_VEnableFusedOps_Serial, 116
- N_VEnableFusedOps_Sycl, 162
- N_VEnableLinearCombination_Cuda, 146
- N_VEnableLinearCombination_Hip, 152
- N_VEnableLinearCombination_ManyVector, 174
- N_VEnableLinearCombination_MPIManyVector, 179
- N_VEnableLinearCombination_OpenMP, 127
- N_VEnableLinearCombination_OpenMPDEV, 168
- N_VEnableLinearCombination_Parallel, 122

- N_VEnableLinearCombination_ParHyp, 137
 N_VEnableLinearCombination_Petsc, 140
 N_VEnableLinearCombination_Pthreads, 133
 N_VEnableLinearCombination_Raja, 158
 N_VEnableLinearCombination_Serial, 117
 N_VEnableLinearCombination_Sycl, 163
 N_VEnableLinearCombinationVectorArray_Cuda, 147
 N_VEnableLinearCombinationVectorArray_Hip, 152
 N_VEnableLinearCombinationVectorArray_OpenMP, 128
 N_VEnableLinearCombinationVectorArray_OpenMPDEV, 169
 N_VEnableLinearCombinationVectorArray_OpenMPParallel, 123
 N_VEnableLinearCombinationVectorArray_OpenMPParHyp, 137
 N_VEnableLinearCombinationVectorArray_OpenMPPetsc, 141
 N_VEnableLinearCombinationVectorArray_OpenMPThreads, 133
 N_VEnableLinearCombinationVectorArray_OpenMPRaja, 158
 N_VEnableLinearCombinationVectorArray_OpenMPSerial, 117
 N_VEnableLinearCombinationVectorArray_OpenMPSycl, 163
 N_VEnableLinearSumVectorArray_Cuda, 146
 N_VEnableLinearSumVectorArray_Hip, 152
 N_VEnableLinearSumVectorArray_ManyVector, 175
 N_VEnableLinearSumVectorArray_MPIManyVector, 180
 N_VEnableLinearSumVectorArray_OpenMP, 128
 N_VEnableLinearSumVectorArray_OpenMPDEV, 169
 N_VEnableLinearSumVectorArray_Parallel, 123
 N_VEnableLinearSumVectorArray_ParHyp, 137
 N_VEnableLinearSumVectorArray_Petsc, 141
 N_VEnableLinearSumVectorArray_Pthreads, 133
 N_VEnableLinearSumVectorArray_Raja, 158
 N_VEnableLinearSumVectorArray_Serial, 117
 N_VEnableLinearSumVectorArray_Sycl, 163
 N_VEnableScaleAddMulti_Cuda, 146
 N_VEnableScaleAddMulti_Hip, 152
 N_VEnableScaleAddMulti_ManyVector, 174
 N_VEnableScaleAddMulti_MPIManyVector, 179
 N_VEnableScaleAddMulti_OpenMP, 128
 N_VEnableScaleAddMulti_OpenMPDEV, 168
 N_VEnableScaleAddMulti_Parallel, 122
 N_VEnableScaleAddMulti_ParHyp, 137
 N_VEnableScaleAddMulti_Petsc, 141
 N_VEnableScaleAddMulti_Pthreads, 133
 N_VEnableScaleAddMulti_Raja, 158
 N_VEnableScaleAddMulti_Serial, 117
 N_VEnableScaleAddMulti_Sycl, 163
 N_VEnableScaleAddMultiVectorArray_Cuda, 147
 N_VEnableScaleAddMultiVectorArray_Hip, 153
 N_VEnableScaleAddMultiVectorArray_OpenMP, 129
 N_VEnableScaleAddMultiVectorArray_OpenMPDEV, 169
 N_VEnableScaleAddMultiVectorArray_Parallel, 124
 N_VEnableScaleAddMultiVectorArray_ParHyp, 138
 N_VEnableScaleAddMultiVectorArray_Petsc, 142
 N_VEnableScaleAddMultiVectorArray_Pthreads, 134
 N_VEnableScaleAddMultiVectorArray_Raja, 159
 N_VEnableScaleAddMultiVectorArray_Serial, 118
 N_VEnableScaleAddMultiVectorArray_Sycl, 163
 N_VEnableScaleVectorArray_Cuda, 146
 N_VEnableScaleVectorArray_Hip, 152
 N_VEnableScaleVectorArray_ManyVector, 175
 N_VEnableScaleVectorArray_MPIManyVector, 180
 N_VEnableScaleVectorArray_OpenMP, 128
 N_VEnableScaleVectorArray_OpenMPDEV, 169
 N_VEnableScaleVectorArray_Parallel, 123
 N_VEnableScaleVectorArray_ParHyp, 137
 N_VEnableScaleVectorArray_Petsc, 141
 N_VEnableScaleVectorArray_Pthreads, 133
 N_VEnableScaleVectorArray_Raja, 158
 N_VEnableScaleVectorArray_Serial, 117
 N_VEnableScaleVectorArray_Sycl, 163
 N_VEnableWrmsNormMaskVectorArray_Cuda, 147
 N_VEnableWrmsNormMaskVectorArray_Hip, 153
 N_VEnableWrmsNormMaskVectorArray_ManyVector, 175
 N_VEnableWrmsNormMaskVectorArray_MPIManyVector, 180
 N_VEnableWrmsNormMaskVectorArray_OpenMP, 129
 N_VEnableWrmsNormMaskVectorArray_OpenMPDEV, 169
 N_VEnableWrmsNormMaskVectorArray_Parallel, 123
 N_VEnableWrmsNormMaskVectorArray_ParHyp, 138
 N_VEnableWrmsNormMaskVectorArray_Petsc, 141
 N_VEnableWrmsNormMaskVectorArray_Pthreads, 134
 N_VEnableWrmsNormMaskVectorArray_Serial, 118
 N_VEnableWrmsNormVectorArray_Cuda, 147
 N_VEnableWrmsNormVectorArray_Hip, 153
 N_VEnableWrmsNormVectorArray_ManyVector, 175
 N_VEnableWrmsNormVectorArray_MPIManyVector, 180
 N_VEnableWrmsNormVectorArray_OpenMP, 128
 N_VEnableWrmsNormVectorArray_OpenMPDEV, 169
 N_VEnableWrmsNormVectorArray_Parallel, 123
 N_VEnableWrmsNormVectorArray_ParHyp, 138
 N_VEnableWrmsNormVectorArray_Petsc, 141
 N_VEnableWrmsNormVectorArray_Pthreads, 134
 N_VEnableWrmsNormVectorArray_Serial, 118
 N_VGetArrayPointer_MPIPlusX, 182

- N_VGetDeviceArrayPointer_Cuda, 143
- N_VGetDeviceArrayPointer_Hip, 150
- N_VGetDeviceArrayPointer_OpenMPDEV, 167
- N_VGetDeviceArrayPointer_Raja, 156
- N_VGetDeviceArrayPointer_Sycl, 161
- N_VGetHostArrayPointer_Cuda, 143
- N_VGetHostArrayPointer_Hip, 150
- N_VGetHostArrayPointer_OpenMPDEV, 167
- N_VGetHostArrayPointer_Raja, 156
- N_VGetHostArrayPointer_Sycl, 161
- N_VGetLocalLength_Parallel, 121
- N_VGetLocalVector_MPIPlusX, 182
- N_VGetNumSubvectors_ManyVector, 173
- N_VGetNumSubvectors_MPIManyVector, 179
- N_VGetSubvector_ManyVector, 173
- N_VGetSubvector_MPIManyVector, 178
- N_VGetSubvectorArrayPointer_ManyVector, 173
- N_VGetSubvectorArrayPointer_MPIManyVector, 178
- N_VGetVector_ParHyp, 136
- N_VGetVector_Petsc, 139
- N_VGetVector_Trilinos, 171
- N_VIsManagedMemory_Cuda, 143
- N_VIsManagedMemory_Hip, 150
- N_VIsManagedMemory_Raja, 156
- N_VIsManagedMemory_Sycl, 162
- N_VMake_Cuda, 144
- N_VMake_Hip, 150
- N_VMake_MPIManyVector, 178
- N_VMake_MPIPlusX, 181
- N_VMake_OpenMP, 126
- N_VMake_OpenMPDEV, 167
- N_VMake_Parallel, 121
- N_VMake_ParHyp, 136
- N_VMake_Petsc, 139
- N_VMake_Pthreads, 131
- N_VMake_Raja, 157
- N_VMake_Serial, 115
- N_VMake_Sycl, 160
- N_VMake_Trilinos, 171
- N_VMakeManaged_Cuda, 144
- N_VMakeManaged_Hip, 151
- N_VMakeManaged_Raja, 157
- N_VMakeManaged_Sycl, 160
- N_VMakeWithManagedAllocator_Cuda, 144
- N_VNew_Cuda, 144
- N_VNew_Hip, 150
- N_VNew_ManyVector, 172
- N_VNew_MPIManyVector, 177, 178
- N_VNew_OpenMP, 126
- N_VNew_OpenMPDEV, 167
- N_VNew_Parallel, 120
- N_VNew_Pthreads, 131
- N_VNew_Raja, 156
- N_VNew_Serial, 115
- N_VNew_Sycl, 160
- N_VNewEmpty, 106
- N_VNewEmpty_Cuda, 144
- N_VNewEmpty_Hip, 150
- N_VNewEmpty_OpenMP, 126
- N_VNewEmpty_OpenMPDEV, 167
- N_VNewEmpty_Parallel, 121
- N_VNewEmpty_ParHyp, 136
- N_VNewEmpty_Petsc, 139
- N_VNewEmpty_Pthreads, 131
- N_VNewEmpty_Raja, 157
- N_VNewEmpty_Serial, 115
- N_VNewEmpty_Sycl, 161
- N_VNewManaged_Cuda, 144
- N_VNewManaged_Hip, 150
- N_VNewManaged_Raja, 157
- N_VNewManaged_Sycl, 160
- N_VNewWithMemHelp_Cuda, 144
- N_VNewWithMemHelp_Raja, 157
- N_VNewWithMemHelp_Sycl, 161
- N_VPrint_Cuda, 145
- N_VPrint_Hip, 151
- N_VPrint_OpenMP, 127
- N_VPrint_OpenMPDEV, 168
- N_VPrint_Parallel, 122
- N_VPrint_ParHyp, 136
- N_VPrint_Petsc, 140
- N_VPrint_Pthreads, 132
- N_VPrint_Raja, 157
- N_VPrint_Serial, 116
- N_VPrint_Sycl, 162
- N_VPrintFile_Cuda, 145
- N_VPrintFile_Hip, 151
- N_VPrintFile_OpenMP, 127
- N_VPrintFile_OpenMPDEV, 168
- N_VPrintFile_Parallel, 122
- N_VPrintFile_ParHyp, 137
- N_VPrintFile_Petsc, 140
- N_VPrintFile_Pthreads, 132
- N_VPrintFile_Raja, 158
- N_VPrintFile_Serial, 116
- N_VPrintFile_Sycl, 162
- N_VSetArrayPointer_MPIPlusX, 182
- N_VSetCudaStream_Cuda, 145
- N_VSetDeviceArrayPointer_Cuda, 143
- N_VSetDeviceArrayPointer_Raja, 156
- N_VSetDeviceArrayPointer_Sycl, 161
- N_VSetHostArrayPointer_Cuda, 143
- N_VSetHostArrayPointer_Raja, 156
- N_VSetHostArrayPointer_Sycl, 161
- N_VSetKernelExecPolicy_Cuda, 145
- N_VSetKernelExecPolicy_Hip, 151
- N_VSetKernelExecPolicy_Sycl, 162
- N_VSetSubvectorArrayPointer_ManyVector, 173

- N_VSetSubvectorArrayPointer_MPIManyVector, 178
- NO_INIT_SETUP, 85
- NO_MIN_EPS, 85
- NO_RES_MON, 85
- nonlinear system
 - definition, 19
- NV_COMM_P, 120
- NV_CONTENT_OMP, 125
- NV_CONTENT_OMPDEV, 166
- NV_CONTENT_P, 119
- NV_CONTENT_PT, 130
- NV_CONTENT_S, 114
- NV_DATA_DEV_OMPDEV, 166
- NV_DATA_HOST_OMPDEV, 166
- NV_DATA_OMP, 125
- NV_DATA_P, 120
- NV_DATA_PT, 130
- NV_DATA_S, 114
- NV_GLOBLENGTH_P, 120
- NV_Ith_OMP, 125
- NV_Ith_P, 120
- NV_Ith_PT, 131
- NV_Ith_S, 115
- NV_LENGTH_OMP, 125
- NV_LENGTH_OMPDEV, 166
- NV_LENGTH_PT, 130
- NV_LENGTH_S, 114
- NV_LOCLENGTH_P, 120
- NV_NUM_THREADS_OMP, 125
- NV_NUM_THREADS_PT, 130
- NV_OWN_DATA_OMP, 125
- NV_OWN_DATA_OMPDEV, 166
- NV_OWN_DATA_P, 120
- NV_OWN_DATA_PT, 130
- NV_OWN_DATA_S, 114
- NVECTOR module, 91
- nvector_openmp_mod, 129
- nvector_pthreads_mod, 135

- optional input
 - generic linear solver interface, 52–54
 - solver, 42–52
- optional output
 - band-block-diagonal preconditioner, 69–70
 - generic linear solver interface, 57–61
 - solver, 55–57
 - version, 54–55

- Picard iteration
 - definition, 23
- portability, 32
 - Fortran, 77
- Preconditioner setup routine
 - use in FKINSOL, 83
- reconditioner solve routine
 - use in FKINSOL, 83
- preconditioning
 - advice on, 29
 - setup and solve phases, 29
 - user-supplied, 53–54, 65, 66
- PRNT_LEVEL, 85
- problem-defining function, 61

- RCONST, 32
- realtype, 32
- RERR_FUNC, 85
- RMON_CONST, 85
- RMON_PARAMS, 85
- ROUT, 85, 86

- SM_COLS_B, 200
- SM_COLS_D, 195
- SM_COLUMN_B, 63, 200
- SM_COLUMN_D, 63, 195
- SM_COLUMN_ELEMENT_B, 63, 200
- SM_COLUMNS_B, 200
- SM_COLUMNS_D, 195
- SM_COLUMNS_S, 207
- SM_CONTENT_B, 198
- SM_CONTENT_D, 194
- SM_CONTENT_S, 205
- SM_DATA_B, 200
- SM_DATA_D, 195
- SM_DATA_S, 207
- SM_ELEMENT_B, 63, 200
- SM_ELEMENT_D, 63, 195
- SM_INDEXPTRS_S, 207
- SM_INDEXVALS_S, 207
- SM_LBAND_B, 200
- SM_LDATA_B, 200
- SM_LDATA_D, 195
- SM_LDIM_B, 200
- SM_NNZ_S, 64, 207
- SM_NP_S, 207
- SM_ROWS_B, 200
- SM_ROWS_D, 195
- SM_ROWS_S, 207
- SM_SPARSETYPE_S, 207
- SM_SUBAND_B, 200
- SM_UBAND_B, 200
- SMALL_REAL, 32
- SSTEP_TOL, 85
- SUNBandMatrix, 35, 201
- SUNBandMatrix_Cols, 202
- SUNBandMatrix_Column, 203
- SUNBandMatrix_Columns, 202
- SUNBandMatrix_Data, 202
- SUNBandMatrix_LDim, 202

- SUNBandMatrix_LowerBandwidth, 202
- SUNBandMatrix_Print, 201
- SUNBandMatrix_Rows, 201
- SUNBandMatrix_StoredUpperBandwidth, 202
- SUNBandMatrix_UpperBandwidth, 202
- SUNBandMatrixStorage, 201
- SUNDenseMatrix, 35, 196
- SUNDenseMatrix_Cols, 196
- SUNDenseMatrix_Column, 197
- SUNDenseMatrix_Columns, 196
- SUNDenseMatrix_Data, 196
- SUNDenseMatrix_LData, 196
- SUNDenseMatrix_Print, 196
- SUNDenseMatrix_Rows, 196
- sundials/sundials_linearsolver.h, 227
- sundials_nvector.h, 33
- sundials_types.h, 32, 33
- SUNDIALSGetVersion, 54
- SUNDIALSGetVersionNumber, 55
- sunindextype, 32
- SUNLinearSolver, 227, 236
- SUNLinearSolver module, 227
- SUNLINEARSOLVER_DIRECT, 63, 229, 239
- SUNLINEARSOLVER_ITERATIVE, 229, 239, 240
- SUNLINEARSOLVER_MATRIX_EMBEDDED, 229, 240
- SUNLINEARSOLVER_MATRIX_ITERATIVE, 229, 239
- sunlinsol/sunlinsol_band.h, 33
- sunlinsol/sunlinsol_dense.h, 33
- sunlinsol/sunlinsol_klu.h, 33
- sunlinsol/sunlinsol_lapackband.h, 33
- sunlinsol/sunlinsol_lapackdense.h, 33
- sunlinsol/sunlinsol_pcg.h, 33
- sunlinsol/sunlinsol_spgmcs.h, 33
- sunlinsol/sunlinsol_spgfmr.h, 33
- sunlinsol/sunlinsol_spgmr.h, 33
- sunlinsol/sunlinsol_sptfqmr.h, 33
- sunlinsol/sunlinsol_superlumt.h, 33
- SUNLinSol_Band, 38, 245
- SUNLinSol_cuSolverSp_batchQR, 268
- SUNLinSol_cuSolverSp_batchQR_GetDescription, 268
- SUNLinSol_cuSolverSp_batchQR_SetDescription, 268
- SUNLinSol_Dense, 38, 243
- SUNLinSol_KLU, 38, 253
- SUNLinSol_KLUReInit, 254
- SUNLinSol_KLUSetOrdering, 257
- SUNLinSol_LapackBand, 38, 250
- SUNLinSol_LapackDense, 38, 248
- SUNLinSol_MagmaDense, 270
- SUNLinSol_OneMklDense, 271
- SUNLinSol_PCG, 38, 301, 304
- SUNLinSol_PCGSetMaxl, 302
- SUNLinSol_PCGSetPrecType, 302
- SUNLinSol_SPBCGS, 38, 287, 291
- SUNLinSol_SPBCGSSetMaxl, 289
- SUNLinSol_SPBCGSSetPrecType, 289
- SUNLinSol_SPGFMR, 38, 280, 283, 284
- SUNLinSol_SPGFMRSetMaxRestarts, 282
- SUNLinSol_SPGFMRSetPrecType, 281
- SUNLinSol_SPGMR, 38, 272, 276
- SUNLinSol_SPGMRSetMaxRestarts, 274
- SUNLinSol_SPGMRSetPrecType, 273, 274
- SUNLinSol_SPTFQMR, 38, 294, 297
- SUNLinSol_SPTFQMRSetMaxl, 295
- SUNLinSol_SPTFQMRSetPrecType, 295
- SUNLinSol_SuperLUDIST, 260
- SUNLinSol_SuperLUDIST_GetBerr, 261
- SUNLinSol_SuperLUDIST_GetGridinfo, 261
- SUNLinSol_SuperLUDIST_GetLUstruct, 261
- SUNLinSol_SuperLUDIST_GetScalePermstruct, 261
- SUNLinSol_SuperLUDIST_GetSOLVEstruct, 262
- SUNLinSol_SuperLUDIST_GetSuperLUOptions, 261
- SUNLinSol_SuperLUDIST_GetSuperLUStat, 262
- SUNLinSol_SuperLUMT, 38, 263
- SUNLinSol_SuperLUMTSetOrdering, 265
- SUNLinSolFree, 36, 228, 231
- SUNLinSolGetID, 228, 229
- SUNLinSolGetType, 228, 229, 240
- SUNLinSolInitialize, 228, 229
- SUNLinSolLastFlag, 233
- SUNLinSolNewEmpty, 238
- SUNLinSolNumIters, 232
- SUNLinSolResNorm, 232
- SUNLinSolSetATimes, 229–231, 240
- SUNLinSolSetInfoFile_PCG, 302
- SUNLinSolSetInfoFile_SPBCGS, 289
- SUNLinSolSetInfoFile_SPGFMR, 282
- SUNLinSolSetInfoFile_SPGMR, 274
- SUNLinSolSetInfoFile_SPTFQMR, 295
- SUNLinSolSetPreconditioner, 231
- SUNLinSolSetPrintLevel_PCG, 303
- SUNLinSolSetPrintLevel_SPBCGS, 290
- SUNLinSolSetPrintLevel_SPGFMR, 282
- SUNLinSolSetPrintLevel_SPGMR, 275
- SUNLinSolSetPrintLevel_SPTFQMR, 296
- SUNLinSolSetScalingVectors, 232
- SUNLinSolSetup, 228, 230, 240
- SUNLinSolSetZeroGuess, 232
- SUNLinSolSolve, 228, 230, 240
- SUNLinSolSpace, 233
- SUNMatCopyOps, 190
- SUNMatDestroy, 36
- SUNMatNewEmpty, 190
- SUNMatrix, 187, 192
- SUNMatrix module, 187
- SUNMatrix_SLUNRloc, 211
- SUNMatrix_SLUNRloc_OwnData, 212

SUNMatrix_SLUNRloc_Print, 212
SUNMatrix_SLUNRloc_ProcessGrid, 212
SUNMatrix_SLUNRloc_SuperMatrix, 212
SUNMemory, 309
SUNMemory module, 309
SUNMemoryHelper, 309
SUNMemoryType, 309
SUNSparseFromBandMatrix, 208
SUNSparseFromDenseMatrix, 208
SUNSparseMatrix, 35, 207
SUNSparseMatrix_Columns, 209
SUNSparseMatrix_Data, 209
SUNSparseMatrix_IndexPointers, 210
SUNSparseMatrix_IndexValues, 210
SUNSparseMatrix_NNZ, 64, 209
SUNSparseMatrix_NP, 209
SUNSparseMatrix_Print, 209
SUNSparseMatrix_Realloc, 208
SUNSparseMatrix_Reallocate, 208
SUNSparseMatrix_Rows, 209
SUNSparseMatrix_SparseType, 209

UNIT_ROUNDOFF, 32

User main program

- FKINBBD usage, 87
- FKINSOL usage, 79
- KINBBDPRE usage, 68
- KINSOL usage, 34

user_data, 61, 67, 68

