User Documentation for KINSOL v6.0.0

SUNDIALS v6.0.0

Alan C. Hindmarsh[1], Radu Serban[1], Cody J. Balos[1],
David J. Gardner[1], Daniel R. Reynolds[2], and Carol S. Woodward[1]

[1]*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory*
[2]*Department of Mathematics, Southern Methodist University*

January 05, 2022

UCRL-SM-208116

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

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.

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

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

Chapter 1

Introduction

KINSOL is part of a software family called SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic equation Solvers [20]. This suite consists of CVODE, ARKODE, KINSOL, and IDA, and variants of these with sensitivity analysis capabilities.

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

1.1 Historical Background

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

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

At present, KINSOL may utilize a variety of Krylov methods provided in SUNDIALS. These methods include the GMRES (Generalized Minimal RESidual) [30], FGMRES (Flexible Generalized Minimum RESidual) [29], Bi-CGStab (Bi-Conjugate Gradient Stabilized) [33], TFQMR (Transpose-Free Quasi-Minimal Residual) [16], and PCG (Preconditioned Conjugate Gradient) [19] 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, 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 options, 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 sparse direct solvers, including KLU [9, 37] and the threaded sparse direct solver, SuperLU_MT [11, 24, 42], among others (see Chapter §8 for further details).

In the process of translating NKSOL into C, the overall KINSOL organization has been changed considerably. One key feature of the KINSOL organization is that a separate module devoted to vector operations was created. This module facilitated extension to multiprosessor environments with minimal impact on the rest of the solver. The vector module design is shared across the SUNDIALS suite. This *N_Vector* module is written in terms of abstract vector operations with the actual routines attached by a particular implementation (such as serial or parallel) of `N_Vector`. This abstraction allows writing the SUNDIALS solvers in a manner independent of the actual `N_Vector` implementation (which can be user-supplied), as well as allowing more than one `N_Vector` module linked into an executable file. SUNDIALS (and thus KINSOL) is supplied with serial, MPI-parallel, OpenMP and Pthreads thread-parallel `N_Vector` implementations, as well as multiple `N_Vector` implementations designed to leverage GPU architectures (see Chapter §6 for further details).

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

1.2.1 Changes in vx.x.x

Additionally export `SUNDIALS::<lib>` targets with no static/shared suffix for use within the build directory (this mirrors how the targets are exported upon installation).

Fixed memory leaks in the SUNLINSOL_SUPERLUMT linear solver.

Added new reduction implementations for the CUDA and HIP NVECTORs that use shared memory (local data storage) instead of atomics. These new implementations are recommended when the target hardware does not provide atomic support for the floating point precision that SUNDIALS is being built with. The HIP vector uses these by default, but the *N_VSetKernelExecPolicy_Cuda()* and *N_VSetKernelExecPolicy_Hip()* functions can be used to choose between different reduction implementations.

CMAKE_C_STANDARD is now set to 99 by default.

Fixed `sundials_export.h` include in `sundials_config.h`.

Fixed exported `SUNDIALSConfig.cmake` when profiling is enabled without Caliper.

1.2.2 Changes in v6.0.0

SUNContext

SUNDIALS v6.0.0 introduces a new *SUNContext* object on which all other SUNDIALS objects depend. As such, the constructors for all SUNDIALS packages, vectors, matrices, linear solvers, nonlinear solvers, and memory helpers have been updated to accept a context as the last input. Users upgrading to SUNDIALS v6.0.0 will need to call *SUNContext_Create()* to create a context object with before calling any other SUNDIALS library function, and then provide this object to other SUNDIALS constructors. The context object has been introduced to allow SUNDIALS to provide new features, such as the profiling/instrumentation also introduced in this release, while maintaining thread-safety. See the documentation section on the *SUNContext* for more details.

A script `upgrade-to-sundials-6-from-5.sh` has been provided with the release (obtainable from the GitHub release page) to help ease the transition to SUNDIALS v6.0.0. The script will add a `SUNCTX_PLACEHOLDER` argument to all of the calls to SUNDIALS constructors that now require a `SUNContext` object. It can also update deprecated SUNDIALS constants/types to the new names. It can be run like this:

```
> ./upgrade-to-sundials-6-from-5.sh <files to update>
```

SUNProfiler

A capability to profile/instrument SUNDIALS library code has been added. This can be enabled with the CMake option *SUNDIALS_BUILD_WITH_PROFILING*. A built-in profiler will be used by default, but the Caliper library can also be used instead with the CMake option *ENABLE_CALIPER*. See the documentation section on profiling for more details. **WARNING**: Profiling will impact performance, and should be enabled judiciously.

SUNMemoryHelper

The *SUNMemoryHelper* functions *SUNMemoryHelper_Alloc()*, *SUNMemoryHelper_Dealloc()*, and *SUNMemoryHelper_Copy()* have been updated to accept an opaque handle as the last input. At a minimum, user-defined *SUNMemoryHelper* implementations will need to update these functions to accept the additional argument. Typically, this handle is the execution stream (e.g., a CUDA/HIP stream or SYCL queue) for the operation. The *CUDA*, *HIP*, and *SYCL* implementations have been updated accordingly. Additionally, the constructor *SUNMemoryHelper_Sycl()* has been updated to remove the SYCL queue as an input.

NVector

Two new optional vector operations, *N_VDotProdMultiLocal()* and *N_VDotProdMultiAllReduce()*, have been added to support low-synchronization methods for Anderson acceleration.

The CUDA, HIP, and SYCL execution policies have been moved from the `sundials` namespace to the `sundials::cuda`, `sundials::hip`, and `sundials::sycl` namespaces respectively. Accordingly, the prefixes "Cuda", "Hip", and "Sycl" have been removed from the execution policy classes and methods.

The `Sundials` namespace used by the Trilinos Tpetra NVector has been replaced with the `sundials::trilinos::nvector_tpetra` namespace.

The serial, PThreads, PETSc, *hypre*, Parallel, OpenMP_DEV, and OpenMP vector functions `N_VCloneVectorArray_*` and `N_VDestroyVectorArray_*` have been deprecated. The generic *N_VCloneVectorArray()* and *N_VDestroyVectorArray()* functions should be used instead.

The previously deprecated constructor `N_VMakeWithManagedAllocator_Cuda` and the function `N_VSetCudaStream_Cuda` have been removed and replaced with *N_VNewWithMemHelp_Cuda()* and `N_VSetKerrnelExecPolicy_Cuda()` respectively.

The previously deprecated macros `PVEC_REAL_MPI_TYPE` and `PVEC_INTEGER_MPI_TYPE` have been removed and replaced with `MPI_SUNREALTYPE` and `MPI_SUNINDEXTYPE` respectively.

SUNLinearSolver

The following previously deprecated functions have been removed:

Removed	Replacement
SUNBandLinearSolver	*SUNLinSol_Band()*
SUNDenseLinearSolver	*SUNLinSol_Dense()*
SUNKLU	*SUNLinSol_KLU()*
SUNKLUReInit	*SUNLinSol_KLUReInit()*
SUNKLUSetOrdering	*SUNLinSol_KLUSetOrdering()*
SUNLapackBand	*SUNLinSol_LapackBand()*
SUNLapackDense	*SUNLinSol_LapackDense()*
SUNPCG	*SUNLinSol_PCG()*
SUNPCGSetPrecType	*SUNLinSol_PCGSetPrecType()*
SUNPCGSetMaxl	*SUNLinSol_PCGSetMaxl()*
SUNSPBCGS	*SUNLinSol_SPBCGS()*
SUNSPBCGSSetPrecType	*SUNLinSol_SPBCGSSetPrecType()*
SUNSPBCGSSetMaxl	*SUNLinSol_SPBCGSSetMaxl()*
SUNSPFGMR	*SUNLinSol_SPFGMR()*
SUNSPFGMRSetPrecType	*SUNLinSol_SPFGMRSetPrecType()*
SUNSPFGMRSetGSType	*SUNLinSol_SPFGMRSetGSType()*
SUNSPFGMRSetMaxRestarts	*SUNLinSol_SPFGMRSetMaxRestarts()*
SUNSPGMR	*SUNLinSol_SPGMR()*
SUNSPGMRSetPrecType	*SUNLinSol_SPGMRSetPrecType()*
SUNSPGMRSetGSType	*SUNLinSol_SPGMRSetGSType()*
SUNSPGMRSetMaxRestarts	*SUNLinSol_SPGMRSetMaxRestarts()*
SUNSPTFQMR	*SUNLinSol_SPTFQMR()*
SUNSPTFQMRSetPrecType	*SUNLinSol_SPTFQMRSetPrecType()*
SUNSPTFQMRSetMaxl	*SUNLinSol_SPTFQMRSetMaxl()*
SUNSuperLUMT	*SUNLinSol_SuperLUMT()*
SUNSuperLUMTSetOrdering	*SUNLinSol_SuperLUMTSetOrdering()*

KINSOL

New orthogonalization methods were added for use within the KINSOL Anderson acceleration routine. See §2.13 and *KINSetOrthAA()* for more details.

The KINSOL Fortran 77 interface has been removed. See §4.4 and the F2003 example programs for more details using the SUNDIALS Fortran 2003 module interfaces.

Deprecations

In addition to the deprecations noted elsewhere, many constants, types, and functions have been renamed so that they are properly namespaced. The old names have been deprecated and will be removed in SUNDIALS v7.0.0.

The following constants, macros, and typedefs are now deprecated:

Deprecated Name	New Name
realtype	sunrealtype
booleantype	sunbooleantype
RCONST	SUN_RCONST
BIG_REAL	SUN_BIG_REAL
SMALL_REAL	SUN_SMALL_REAL
UNIT_ROUNDOFF	SUN_UNIT_ROUNDOFF
PREC_NONE	SUN_PREC_NONE
PREC_LEFT	SUN_PREC_LEFT
PREC_RIGHT	SUN_PREC_RIGHT
PREC_BOTH	SUN_PREC_BOTH
MODIFIED_GS	SUN_MODIFIED_GS
CLASSICAL_GS	SUN_CLASSICAL_GS
ATimesFn	SUNATimesFn
PSetupFn	SUNPSetupFn
PSolveFn	SUNPSolveFn
DlsMat	SUNDlsMat
DENSE_COL	SUNDLS_DENSE_COL
DENSE_ELEM	SUNDLS_DENSE_ELEM
BAND_COL	SUNDLS_BAND_COL
BAND_COL_ELEM	SUNDLS_BAND_COL_ELEM
BAND_ELEM	SUNDLS_BAND_ELEM

In addition, the following functions are now deprecated (compile-time warnings will be thrown if supported by the compiler):

Deprecated Name	New Name
KINDlsSetLinearSolver	KINSetLinearSolver
KINDlsSetJacFn	KINSetJacFn
KINDlsGetWorkSpace	KINGetLinWorkSpace
KINDlsGetNumJacEvals	KINGetNumJacEvals
KINDlsGetNumFuncEvals	KINGetNumLinFuncEvals
KINDlsGetLastFlag	KINGetLastLinFlag
KINDlsGetReturnFlagName	KINGetLinReturnFlagName
KINSpilsSetLinearSolver	KINSetLinearSolver
KINSpilsSetPreconditioner	KINSetPreconditioner
KINSpilsSetJacTimesVecFn	KINSetJacTimesVecFn
KINSpilsGetWorkSpace	KINGetLinWorkSpace
KINSpilsGetNumPrecEvals	KINGetNumPrecEvals
KINSpilsGetNumPrecSolves	KINGetNumPrecSolves
KINSpilsGetNumLinIters	KINGetNumLinIters
KINSpilsGetNumConvFails	KINGetNumLinConvFails
KINSpilsGetNumJtimesEvals	KINGetNumJtimesEvals
KINSpilsGetNumFuncEvals	KINGetNumLinFuncEvals
KINSpilsGetLastFlag	KINGetLastLinFlag
KINSpilsGetReturnFlagName	KINGetLinReturnFlagName
DenseGETRF	SUNDlsMat_DenseGETRF
DenseGETRS	SUNDlsMat_DenseGETRS
denseGETRF	SUNDlsMat_denseGETRF
denseGETRS	SUNDlsMat_denseGETRS

continues on next page

Table 1.1 – continued from previous page

Deprecated Name	New Name
DensePOTRF	SUNDlsMat_DensePOTRF
DensePOTRS	SUNDlsMat_DensePOTRS
densePOTRF	SUNDlsMat_densePOTRF
densePOTRS	SUNDlsMat_densePOTRS
DenseGEQRF	SUNDlsMat_DenseGEQRF
DenseORMQR	SUNDlsMat_DenseORMQR
denseGEQRF	SUNDlsMat_denseGEQRF
denseORMQR	SUNDlsMat_denseORMQR
DenseCopy	SUNDlsMat_DenseCopy
denseCopy	SUNDlsMat_denseCopy
DenseScale	SUNDlsMat_DenseScale
denseScale	SUNDlsMat_denseScale
denseAddIdentity	SUNDlsMat_denseAddIdentity
DenseMatvec	SUNDlsMat_DenseMatvec
denseMatvec	SUNDlsMat_denseMatvec
BandGBTRF	SUNDlsMat_BandGBTRF
bandGBTRF	SUNDlsMat_bandGBTRF
BandGBTRS	SUNDlsMat_BandGBTRS
bandGBTRS	SUNDlsMat_bandGBTRS
BandCopy	SUNDlsMat_BandCopy
bandCopy	SUNDlsMat_bandCopy
BandScale	SUNDlsMat_BandScale
bandScale	SUNDlsMat_bandScale
bandAddIdentity	SUNDlsMat_bandAddIdentity
BandMatvec	SUNDlsMat_BandMatvec
bandMatvec	SUNDlsMat_bandMatvec
ModifiedGS	SUNModifiedGS
ClassicalGS	SUNClassicalGS
QRfact	SUNQRFact
QRsol	SUNQRsol
DlsMat_NewDenseMat	SUNDlsMat_NewDenseMat
DlsMat_NewBandMat	SUNDlsMat_NewBandMat
DestroyMat	SUNDlsMat_DestroyMat
NewIntArray	SUNDlsMat_NewIntArray
NewIndexArray	SUNDlsMat_NewIndexArray
NewRealArray	SUNDlsMat_NewRealArray
DestroyArray	SUNDlsMat_DestroyArray
AddIdentity	SUNDlsMat_AddIdentity
SetToZero	SUNDlsMat_SetToZero
PrintMat	SUNDlsMat_PrintMat
newDenseMat	SUNDlsMat_newDenseMat
newBandMat	SUNDlsMat_newBandMat
destroyMat	SUNDlsMat_destroyMat
newIntArray	SUNDlsMat_newIntArray
newIndexArray	SUNDlsMat_newIndexArray
newRealArray	SUNDlsMat_newRealArray
destroyArray	SUNDlsMat_destroyArray

In addition, the entire `sundials_lapack.h` header file is now deprecated for removal in SUNDIALS v7.0.0. Note, this header file is not needed to use the SUNDIALS LAPACK linear solvers.

1.2.3 Changes in v5.8.0

The RAJA `N_Vector` 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 `SUNLinearSolver` 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 §8.9 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.

1.2.4 Changes in v5.7.0

A new `N_Vector` 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 §6.13 for more details. This module is considered experimental and is subject to major changes even in minor releases.

A new `SUNMatrix` and `SUNLinearSolver` 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 §8.8 for more details.

1.2.5 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.

1.2.6 Changes in v5.6.0

A new N_Vector implementation based on the AMD ROCm HIP platform has been added. This vector can target NVIDIA or AMD GPUs. See §6.11 for more details. This module is considered experimental and is subject to change from version to version.

The RAJA N_Vector 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_CUSOLVERSP_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.

1.2.7 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.

1.2.8 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 §4.5.1, §6.10, §6.12, and §9 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.

1.2.9 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 perfomance 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 capabilties.

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

1.2.10 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.

1.2.11 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 §2 and the description of the KINSetDampingAA function for more details.

1.2.12 Changes in v5.0.0

1.2.12.1 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.

1.2.12.2 NVECTOR module changes

- Two new functions were added to aid in creating custom `N_Vector` objects. The constructor `N_VNewEmpty` allocates an "empty" generic `N_Vector` 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 `N_Vector` 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 `N_Vector` API by ensuring all operations are copied when cloning objects. See §6.1.1 for more details.
- Two new `N_Vector` 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 §6.16 and §6.17 for more details.
- One new required vector operation and ten new optional vector operations have been added to the `N_Vector` 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 `N_Vector` implementation defines any of the local operations as `NULL`, then the `NVECTOR_MPIMANYVECTOR` will call standard `N_Vector` operations to complete the computation. See §6.2.4 for more details.
- An additional `N_Vector` 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 §6.18 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 §6.10.1 for more details.
- Added new Fortran 2003 interfaces for most `N_Vector` modules. See Chapter §6 for more details on how to use the interfaces.
- Added three new `N_Vector` utility functions, `FN_VGetVecAtIndexVectorArray`, `FN_VSetVecAtIndexVectorArray`, and `FN_VNewVectorArray`, for working with `N_Vector` arrays when using the Fortran 2003 interfaces. See §6.1.1 for more details.

1.2.12.3 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 §7.1 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 §7.2 for more details.
- The generic `SUNMatrix` API now defines error codes to be returned by `SUNMatrix` operations. Operations which return an integer flag indiciating success/failure may return different values than previously. See §7.2.1 for more details.
- A new `SUNMatrix` (and `SUNLinearSolver`) implementation was added to facilitate the use of the SuperLU_DIST library with SUNDIALS. See §7.9 for more details.
- Added new Fortran 2003 interfaces for most `SUNMatrix` modules. See Chapter §7 for more details on how to use the interfaces.

1.2.12.4 SUNLinearSolver module changes

- A new function was added to aid in creating custom `SUNLinearSolver` objects. The constructor `SUNLinSolNewEmpty` allocates an "empty" generic `SUNLinearSolver` 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 `SUNLinearSolver` API by ensuring only required operations need to be set. See §8.1.8 for more details.
- The return type of the `SUNLinearSolver` 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 `SUNLinearSolver` API, `SUNLinSolGetID`, that returns a `SUNLinearSolver_ID` for identifying the linear solver module.
- The `SUNLinearSolver` API has been updated to make the initialize and setup functions optional.
- A new `SUNLinearSolver` (and `SUNMatrix`) implementation was added to facilitate the use of the SuperLU_DIST library with SUNDIALS. See §8.15 for more details.
- Added a new `SUNLinearSolver` implementation, `SUNLinearSolver_cuSolverSp_batchQR`, which leverages the NVIDIA cuSOLVER sparse batched QR method for efficiently solving block diagonal linear systems on NVIDIA GPUs. See §8.17 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 §8.5.1 for more details.
- Added new Fortran 2003 interfaces for most `SUNLinearSolver` modules. See Chapter §8 for more details on how to use the interfaces.

1.2.12.5 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 §4.4 for more details.

1.2.13 Changes in v4.1.0

An additional `N_Vector` 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 `N_Vector` 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`.

1.2.14 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`.

1.2.15 Changes in v4.0.1

No changes were made in this release.

1.2.16 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 `SUNLinearSolver` 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 `SUNLinearSolver` implementations are included in Chapter §8. 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 `SUNLinearSolver` 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_SPFGMR, 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 N_Vector API. These *optional* operations are disabled by default and may be activated by calling vector specific routines after creating an N_Vector (see Chapter §6 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 N_Vector implementation defines any of these operations as NULL, then standard N_Vector 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 N_Vector implementation for leveraging OpenMP 4.5+ device offloading has been added, NVECTOR_OPENMPDEV. See §6.14 for more details.

1.2.17 Changes in v3.2.1

The changes in this minor release include the following:

- Fixed a bug in the CUDA N_Vector 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 modified.

1.2.18 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 `N_Vector` 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 have been depreated. 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.

1.2.19 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`.

1.2.20 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 `N_Vector` and `SUNMatrix` header files.
- Fixed an indexing bug in the CUDA `N_Vector` implementation of `N_VWrmsNormMask` and revised the RAJA `N_Vector` 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 `SUNLinearSolver` 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.

1.2.21 Changes in v3.1.0

Added `N_Vector` 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.

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

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

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

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

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

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

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

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

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

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

1.2.23 Changes in v2.9.0

Two additional N_Vector 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 N_Vector module now includes a function, N_VGetVectorID, that returns the N_Vector module name.

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

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

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

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

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

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

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

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

1.2.24 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 N_Vector module. As part of these additions, a sparse matrix (CSC format) structure was added to KINSOL. Finally, a variation of GMRES called Flexible GMRES was added.

Otherwise, only relatively minor modifications were made to KINSOL:

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

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

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

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

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

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

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

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

Two new N_Vector 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.

1.2.25 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.

1.2.26 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.

1.2.27 Changes in v2.5.0

The main changes in this release involve a rearrangement of the entire SUNDIALS source tree (see §3). 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.

1.2.28 Changes in v2.4.0

KINSPBCG, KINSPTFQMR, KINDENSE, and KINBAND modules have been added to interface with the Scaled Preconditioned Bi-CGStab (SPBCG), Scaled Preconditioned Transpose-Free Quasi-Minimal Residual (SPTFQMR), DENSE, and BAND linear solver modules, respectively. (For details see Chapter :numref:KINSOL.Usage.CC.) 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 §10.

1.2.29 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.

1.2.30 Changes in v2.2.1

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

1.2.31 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 :numref:KINSOL.Usage.CC.

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 :numref:KINSOL.Usage.CC through §5 only, and looking at examples in [8]. In a different direction, a more expert user with a nonlinear system may want to (a) use a package preconditioner (§5.7), (b) supply his/her own Jacobian or preconditioner routines (§5.6), (c) supply a new `N_Vector` module (Chapter §6), or even (d) supply a different linear solver module (§5.5.2 and Chapter §8).

The structure of this document is as follows:

- In Chapter §2, we provide short descriptions of the numerical methods implemented by KINSOL for the solution of nonlinear systems.
- The following chapter describes the structure of the SUNDIALS suite of solvers (§3) and the software organization of the KINSOL solver (§3.1).
- Chapter :numref:KINSOL.Usage.CC 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.
- Chapter §6 gives a brief overview of the generic `N_Vector` module shared among the various components of SUNDIALS, and details on the four `N_Vector` implementations provided with SUNDIALS.
- Chapter §7 gives a brief overview of the generic `SUNMatrix` module shared among the various components of SUNDIALS, and details on the `SUNMatrix` implementations provided with SUNDIALS.
- Chapter §8 gives a brief overview of the generic `SUNLinearSolver` module shared among the various components of SUNDIALS. This chapter contains details on the `SUNLinearSolver` implementations provided with SUNDIALS. The chapter also contains details on the `SUNLinearSolver` 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 §10), as well as a list of all the constants used for input to and output from KINSOL functions (Appendix §11).

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

1.4 SUNDIALS License and Notices

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.

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

1.4.1 BSD 3-Clause License

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.

1.4.3 SUNDIALS Release Numbers

LLNL-CODE-667205 (ARKODE)

UCRL-CODE-155951 (CVODE)

UCRL-CODE-155950 (CVODES)

UCRL-CODE-155952 (IDA)

UCRL-CODE-237203 (IDAS)

LLNL-CODE-665877 (KINSOL)

1.5 Acknowledgments

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

Chapter 2

Mathematical Considerations

KINSOL solves nonlinear algebraic systems in real N-space.

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

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

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

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

given an initial guess u_0.

2.1 Basic Newton iteration

Depending on the linear solver used, KINSOL can employ either an Inexact Newton method [4, 6, 10, 12, 23], or a Modified Newton method. At the highest level, KINSOL implements the following iteration scheme:

1. Set $u_0 =$ an initial guess
2. For $n = 0, 1, 2, ...$ until convergence do:
   a. Solve $J(u_n)\delta_n = -F(u_n)$
   b. Set $u_{n+1} = u_n + \lambda\delta_n, 0 < \lambda \leq 1$
   c. Test for convergence

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

2.2 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 to various libraries, including KLU [9, 37], SuperLU_MT [11, 24, 42], SuperLU_-Dist [17, 25, 26, 41], and cuSPARSE [40] [Note that users will need to download and install the relevant external 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,
- SPBCG, 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 *2a* is solved exactly, thus resulting in a Modified Newton method (the Jacobian matrix is normally out of date; see below). Note that KINSOL allows the user to enforce a Jacobian evaluation at each iteration thus allowing for an Exact Newton iteration. Note that each direct linear solver is only compatible with a subset of vector representations (see §8.1.7 for details).

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.

2.3 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
- $\|\lambda\delta_n\|_{D_u,\infty} <$ steptol with outdated Jacobian or preconditioner information,

where the norm $\|\cdot\|_{D_u,\infty}$ is defined below in (2.3).

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.

2.4 Scaling

To address the case of ill-conditioned nonlinear systems, KINSOL allows users to prescribe 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. Based on these scaling matrices, we define the following scaled norms:

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

where $\|\cdot\|_\infty$ is the max norm. When scaling values are provided for the solution vector, these values are automatically incorporated into the calculation of the perturbations used for the default difference quotient approximations for Jacobian information; see (2.6) and (2.8) below.

2.5 Globalization strategy

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

KINSOL implements a backtracking algorithm to first find a 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

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

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

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

2.6 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.

2.7 Additional constraints

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

2.8 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

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

with ρ defined as

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

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`.

2.9 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 [14]. 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 [14].

2.10 Difference quotient Jacobian approximations

With the *SUNMATRIX_DENSE* and *SUNMATRIX_BAND* matrix modules, the Jacobian may be supplied by a user routine, or approximated by difference quotients, at the user's option. In the latter case, we use the usual approximation

$$J^{ij} = [F^i(u + \sigma_j e^j) - F^i(u)]/\sigma_j\,. \tag{2.5}$$

The increments σ_j are given by

$$\sigma_j = \sqrt{U}\ \max\left\{|u^j|, 1/D_u^j\right\}\,. \tag{2.6}$$

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

$$J(u)v \approx [F(u + \sigma v) - F(u)]/\sigma\,, \tag{2.7}$$

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

$$\sigma = \frac{\max\{|u^T v|, u_{typ}^T |v|\}}{\|v\|_2^2} \text{sign}(u^T v)\sqrt{U}\,, \tag{2.8}$$

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

2.11 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 $n = 0, 1, 2, ...$ until convergence do:
   - Set $u_{n+1} = (1 - \beta)u_n + \beta G(u_n)$.
   - 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 $n = 0, 1, 2, ...$ until convergence do:
   - Set $u_{n+1} = (1 - \beta)u_n + \beta G(u_n)$ where $G(u_n) \equiv u_n - L^{-1}F(u_n)$.
   - 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 page 182 of [28].

2.12 Anderson Acceleration

The Picard and fixed point methods can be significantly accelerated using Anderson's method [1, 15, 27, 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 $n = 1, 2, ...$ until convergence do:
   a. Set $m_n = \min\{m, n\}$
   b. Set $F_n = (f_{n-m_n}, \ldots, f_n)$, where $f_i = G(u_i) - u_i$

c. Determine $\alpha^{(n)} = (\alpha_0^{(n)}, \ldots, \alpha_{m_n}^{(n)})$ that solves $\min_{\alpha} \|F_n \alpha^T\|_2$ such that $\sum_{i=0}^{m_n} \alpha_i = 1$

d. Set $u_{n+1} = \beta \sum_{i=0}^{m_n} \alpha_i^{(n)} G(u_{n-m_n+i}) + (1-\beta) \sum_{i=0}^{m_n} \alpha_i^{(n)} u_{n-m_n+i}$

e. Test for convergence

It has been implemented in KINSOL by turning the constrained linear least-squares problem in step 3c 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 $n = 1, 2, ...$ until convergence do:
   a. Set $m_n = \min\{m, n\}$
   b. Set $\Delta F_n = (\Delta f_{n-m_n}, \ldots, \Delta f_{n-1})$, where $\Delta f_i = f_{i+1} - f_i$ and $f_i = G(u_i) - u_i$
   c. Determine $\gamma^{(n)} = (\gamma_0^{(n)}, \ldots, \gamma_{m_n-1}^{(n)})$ that solves $\min_{\gamma} \|f_n - \Delta F_n \gamma^T\|_2$
   d. Set $u_{n+1} = G(u_n) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta g_{n-m_n+i} - (1-\beta)(f(u_n) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta f_{n-m_n+i})$ with $\Delta g_i = G(u_{i+1}) - G(u_i)$
   e. Test for convergence

The least-squares problem in 3c 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.

2.13 Anderson Acceleration QR Factorization

The default QR factorization routine used in Anderson acceleration is Modified Gram-Schmidt, a stable orthogonalization routine that requires an increasing number of synchronizations per iteration dependent upon the number of vectors being orthgonalized against. While practical use of Anderson acceleration only requires a small number of vectors to be used in the QR factorization, this linearly scaling number of synchronizations per iteration can yield poor performance when Anderson acceleration is performed in a parallel setting. To combat this poor performance, low synchronization QR routines are available to the user, in particular: Inverse Compact WY Modified Gram-Schmidt [31], along with variants of Classical Gram-Schmidt with Reorthogonalization [18]. While all of these QR factorization routines are mathematically equivalent, they do not exhibit the same stability when performed with floating point arithmetic or in a parallel setting.

Inverse Compact WY Modified Gram-Schmidt, which is based on triangular solve variants of Gram-Schmidt that were developed within the context of GMRES, is an option that only requires two synchronizations per iteration. Additionally, it adds a lower triangular solve at every iteration, but this generally does not affect performance due to the system solve being small i.e., the number of vectors being orthgonalized against.

The remaining orthogonalization options are based on and include Classical Gram-Schmidt with Reorthogonalization (CGS-2). CGS-2 only requires three synchronizations per iteration, but does not exhibit the same stability as Modified Gram-Schmidt. Classical Gram-Schmidt with Delayed Reorthogolonization has the same stability as CGS-2, but it reduces the number of synchronizations per iteration to two.

2.14 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.

2.15 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

SUNDIALS consists of the solvers CVODE and ARKODE for ordinary differential equation (ODE) systems, IDA for differential-algebraic (DAE) systems, and KINSOL for nonlinear 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 following is a list summarizes the basic functionality of each SUNDIALS package:

- CVODE, a solver for stiff and nonstiff ODE systems $\dot{y} = f(t, y)$ 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 $M(t)\,\dot{y} = f_1(t, y)+f_2(t, y)$ 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 $F(u) = 0$.

The various packages in the suite share many common components and are organized as a family. Fig. 3.1 gives a high-level overview of solver packages, the shared vector, matrix, linear solver, and nonlinear solver interfaces (abstract base classes), and the corresponding class implementations provided with SUNDIALS. For classes that provide interfaces to third-party libraries (i.e., LAPACK, KLU, SuperLU_MT, SuperLU_DIST, *hypre*, PETSc, Trilinos, and Raja) users will need to download and compile those packages independently of SUNDIALS. The directory structure is shown in Fig. 3.2.

3.1 KINSOL organization

The KINSOL package is written in ANSI C. The following 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 Fig. 3.3. KINSOL utilizes generic linear solvers defined by the `SUNLinearSolver` (see §8). As such, KINSOL has no knowledge of the method being used to solve the linear and nonlinear systems that arise. For any given user problem, there exists a single nonlinear solver interface and, if necessary, one of the linear system solver interfaces is specified, and invoked as needed during the integration.

KINSOL has a single unified linear solver interface, KINSOLLS, supporting both direct and iterative linear solvers built using the generic `SUNLinearSolver` interface (see §8). These solvers may utilize a `SUNMatrix` object (see §7) for storing Jacobian information, or they may be matrix-free. Since KINSOL can operate on any valid `SUNLinearSolver`,

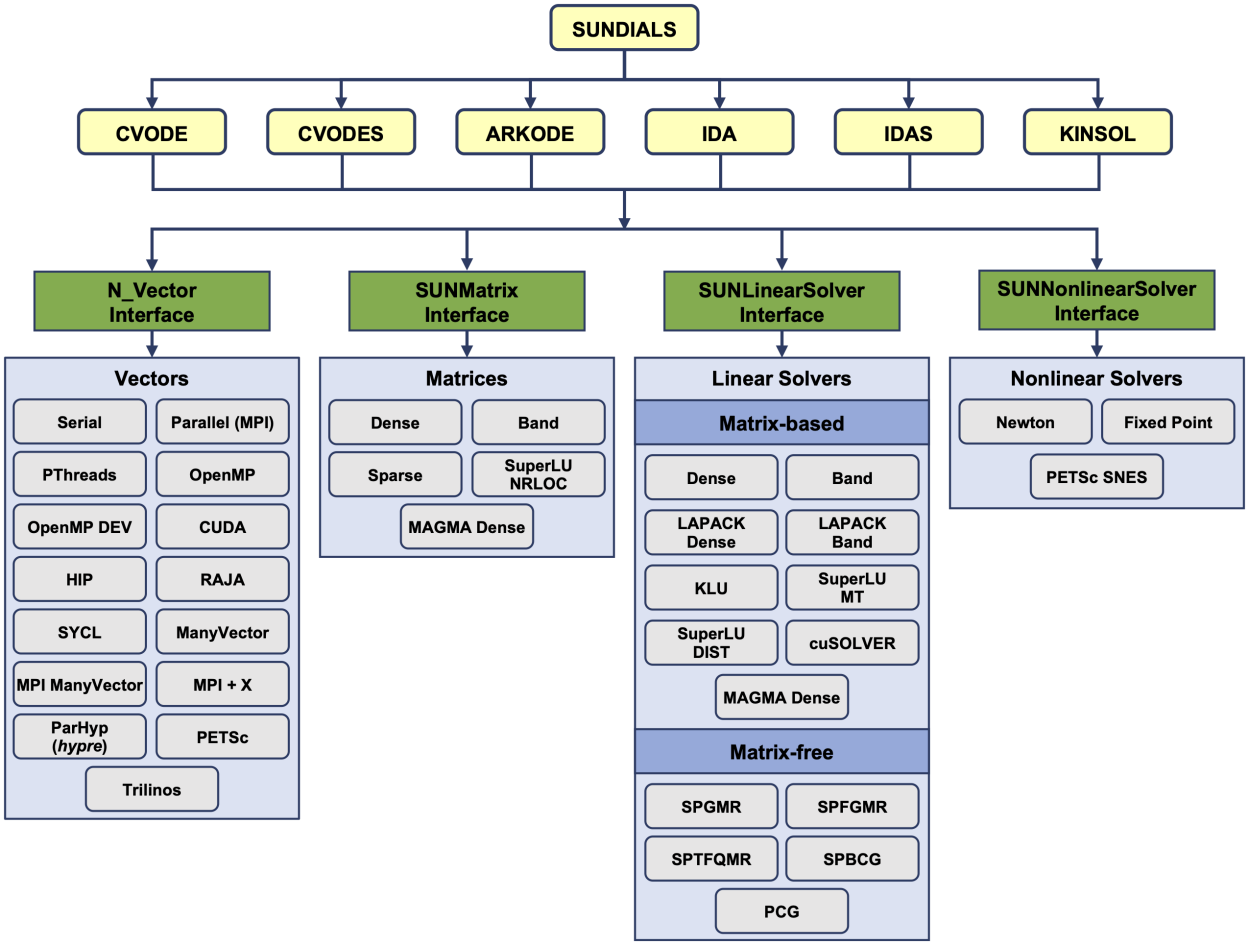

Fig. 3.1: High-level diagram of the SUNDIALS suite.

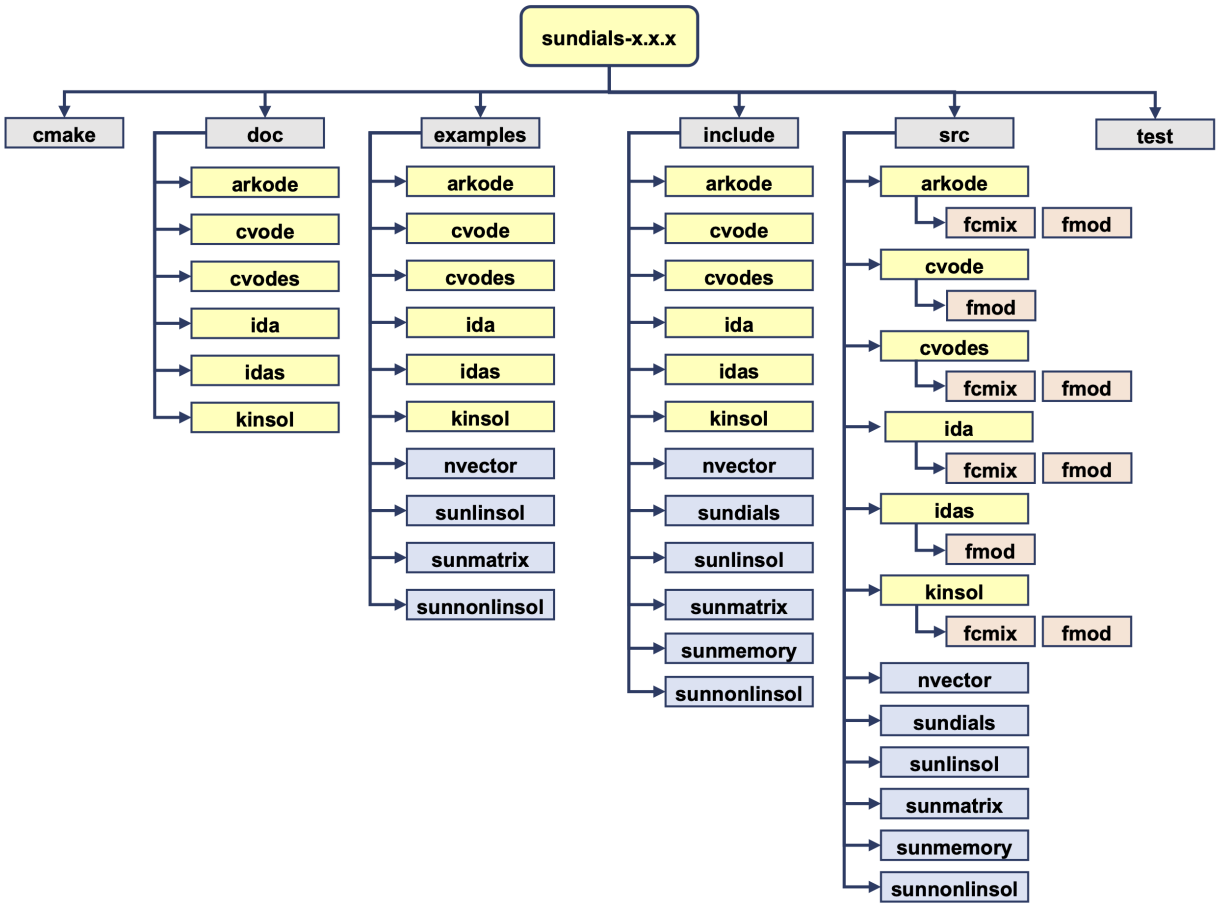

Fig. 3.2: Directory structure of the SUNDIALS source tree.

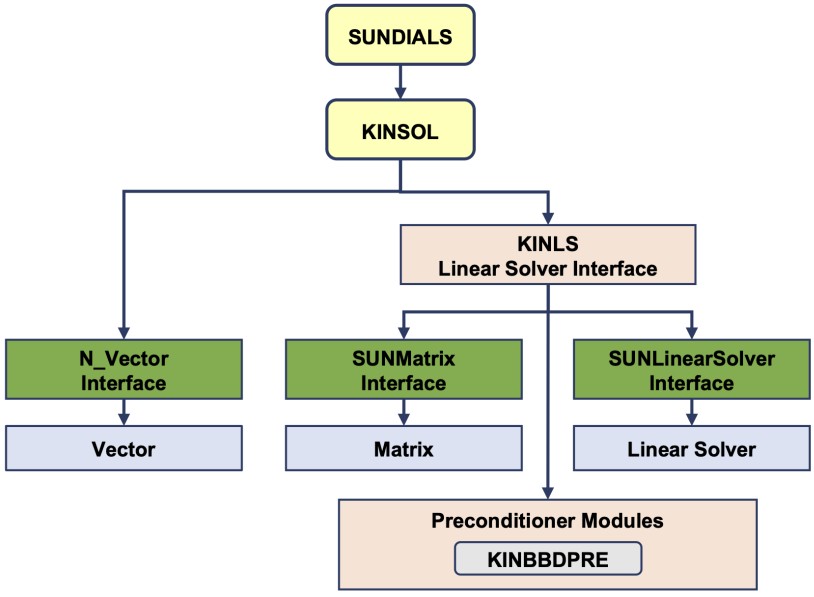

Fig. 3.3: Overall structure diagram of the KINSOL package. Components specific to KINSOL begin with "KINSOL" (KINLS and KINSOLBBDPRE), all other items correspond to generic SUNDIALS vector, matrix, and solver interfaces.

the set of linear solver modules available to KINSOL will expand as new `SUNLinearSolver` implementations are developed.

For users employing *SUNMATRIX_DENSE* or *SUNMATRIX_BAND* Jacobian matrices, KINSOL includes algorithms for their approximation through difference quotients, although the user also has the option of supplying a routine to compute 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, KINSOL includes an algorithm for the approximation by difference quotients of the product Jv. Again, the user has the option of providing routines for this operation, in two phases: setup (preprocessing of Jacobian data) and multiplication.

For preconditioned iterative methods, the preconditioning must be supplied by the user, again in two phases: setup and solve. While there is no default choice of preconditioner analogous to the difference-quotient approximation in the direct case, the references [5, 7], together with the example and demonstration programs included with 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 integration, and only as required to achieve convergence. The call list within the central KINSOL module to each of the four associated functions is fixed, thus allowing the central module to be completely independent of the linear system method.

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

All state information used by KINSOL to solve a given problem is stored in `N_Vector` instances. There is no global data in the KINSOL package, and so, in this respect, it is reentrant. State information specific to the linear and nonlinear solver are saved in the `SUNLinearSolver` and `SUNNonlinearSolver` instances respectively. The reentrancy of KINSOL enables the setting where two or more problems are solved by intermixed or parallel calls to different instances of the package from within a single user program.

Chapter 4

Using SUNDIALS

As discussed in §3, the six solvers packages (CVODE(S), IDA(S), ARKODE, KINSOL) that make up SUNDIALS are built upon common classes/modules for vectors, matrices, and algebraic solvers. In addition, the six packages all leverage some other common infrastructure, which we discuss in this section.

4.1 The SUNContext Type

In SUNDIALS v6.0.0, the concept of a SUNDIALS simulation context was introduced, in particular the `SUNContext` class. All of the SUNDIALS objects (vectors, linear and nonlinear solvers, matrices, etc) that collectively form a SUNDIALS simulation, hold a reference to a common `SUNContext` object.

The `SUNContext` class/type is defined in the header file `sundials/sundials_context.h` as

typedef struct _SUNContext ***SUNContext**

Users should create a `SUNContext` object prior to any other calls to SUNDIALS library functions by calling:

int **SUNContext_Create**(void *comm, *SUNContext* *ctx)

Creates a `SUNContext` object associated with the thread of execution. The data of the `SUNContext` class is private.

Arguments:

- `comm` – a pointer to the MPI communicator or NULL if not using MPI.
- `ctx` – [in,out] upon successful exit, a pointer to the newly created `SUNContext` object.

Returns:

- Will return < 0 if an error occurs, and zero otherwise.

The created `SUNContext` object should be provided to the constructor routines for different SUNDIALS classes/modules. E.g.,

```
SUNContext sunctx;
void* package_mem;
N_Vector x;

SUNContext_Create(NULL, &sunctx);

package_mem = CVodeCreate(..., sunctx);
```

(continues on next page)

(continued from previous page)

```
package_mem = IDACreate(..., sunctx);
package_mem = KINCreate(..., sunctx);
package_mem = ARKStepCreate(..., sunctx);

x = N_VNew_<SomeVector>(..., sunctx);
```

After all other SUNDIALS code, the `SUNContext` object should be freed with a call to:

int **SUNContext_Free**(*SUNContext* *ctx)
Frees the `SUNContext` object.

Arguments:

- `ctx` – pointer to a valid `SUNContext` object, `NULL` upon successful return.

Returns:

- Will return < 0 if an error occurs, and zero otherwise.

Warning: When MPI is being used, the *SUNContext_Free()* must be called prior to `MPI_Finalize`.

The `SUNContext` API further consists of the following functions:

int **SUNContext_GetProfiler**(*SUNContext* ctx, *SUNProfiler* *profiler)
Gets the `SUNProfiler` object associated with the `SUNContext` object.

Arguments:

- `ctx` – a valid `SUNContext` object.
- `profiler` – [in,out] a pointer to the `SUNProfiler` object associated with this context; will be `NULL` if profiling is not enabled.

Returns:

- Will return < 0 if an error occurs, and zero otherwise.

int **SUNContext_SetProfiler**(*SUNContext* ctx, *SUNProfiler* profiler)
Sets the `SUNProfiler` object associated with the `SUNContext` object.

Arguments:

- `ctx` – a valid `SUNContext` object.
- `profiler` – a `SUNProfiler` object to associate with this context; this is ignored if profiling is not enabled.

Returns:

- Will return < 0 if an error occurs, and zero otherwise.

4.1.1 Implications for task-based programming and multi-threading

Applications that need to have *concurrently initialized* SUNDIALS simulations need to take care to understand the following:

#. A `SUNContext` object must only be associated with *one* SUNDIALS simulation (a solver object and its associated vectors etc.) at a time.

- Concurrently initialized is not the same as concurrently executing. Even if two SUNDIALS simulations execute sequentially, if both are initialized at the same time with the same `SUNContext`, behavior is undefined.
- It is OK to reuse a `SUNContext` object with another SUNDIALS simulation after the first simulation has completed and all of the simulation's associated objects (vectors, matrices, algebraic solvers, etc.) have been destroyed.

#. The creation and destruction of a `SUNContext` object is cheap, especially in comparison to the cost of creating/destroying a SUNDIALS solver object.

The following (incomplete) code examples demonstrate these points using CVODE as the example SUNDIALS package.

```
SUNContext sunctxs[num_threads];
int cvode_initialized[num_threads];
void* cvode_mem[num_threads];

// Create
for (int i = 0; i < num_threads; i++) {
   sunctxs[i] = SUNContext_Create(...);
   cvode_mem[i] = CVodeCreate(..., sunctxs[i]);
   cvode_initialized[i] = 0; // not yet initialized
   // set optional cvode inputs...
}

// Solve
#pragma omp parallel for
for (int i = 0; i < num_problems; i++) {
   int retval = 0;
   int tid = omp_get_thread_num();
   if (!cvode_initialized[tid]) {
      retval = CVodeInit(cvode_mem[tid], ...);
      cvode_initialized[tid] = 1;
   } else {
      retval = CVodeReInit(cvode_mem[tid], ...);
   }
   CVode(cvode_mem[i], ...);
}

// Destroy
for (int i = 0; i < num_threads; i++) {
   // get optional cvode outputs...
   CVodeFree(&cvode_mem[i]);
   SUNContext_Free(&sunctxs[i]);
}
```

Since each thread has its own unique CVODE and SUNContext object pair, there should be no thread-safety issues. Users should be sure that you apply the same idea to the other SUNDIALS objects needed as well (e.g. an `N_Vector`).

The variation of the above code example demonstrates another possible approach:

```
// Create, Solve, Destroy
#pragma omp parallel for
for (int i = 0; i < num_problems; i++) {
   int retval = 0;
   void* cvode_mem;
   SUNContext sunctx;

   sunctx = SUNContext_Create(...);
   cvode_mem = CVodeCreate(..., sunctx);
   retval = CVodeInit(cvode_mem, ...);

   // set optional cvode inputs...

   CVode(cvode_mem, ...);

   // get optional cvode outputs...

   CVodeFree(&cvode_mem);
   SUNContext_Free(&sunctx);
}
```

So long as the overhead of creating/destroying the CVODE object is small compared to the cost of solving the ODE, this approach is a fine alternative to the first approach since *SUNContext_Create()* and *SUNContext_Free()* are much cheaper than the CVODE create/free routines.

4.1.2 Convenience class for C++ Users

For C++ users, a class, `sundials::Context`, that follows RAII is provided:

```
namespace sundials
{

class Context
{
public:
   Context(void* comm = NULL)
   {
      SUNContext_Create(comm, &sunctx_);
   }

   operator SUNContext() { return sunctx_; }

   ~Context()
   {
      SUNContext_Free(&sunctx_);
   }

private:
   SUNContext sunctx_;

};
```

(continues on next page)

(continued from previous page)

```
} // namespace sundials
```

4.2 Performance Profiling

SUNDIALS includes a lightweight performance profiling layer that can be enabled at compile-time. Optionally, this profiling layer can leverage Caliper [3] for more advanced instrumentation and profiling. By default, only SUNDIALS library code is profiled. However, a public profiling API can be utilized to leverage the SUNDIALS profiler to time user code regions as well (see §4.2.2).

4.2.1 Enabling Profiling

To enable profiling, SUNDIALS must be built with the CMake option *SUNDIALS_BUILD_WITH_PROFILING* set to `ON`. To utilize Caliper support, the CMake option *ENABLE_CALIPER* must also be set to `ON`. More details in regards to configuring SUNDIALS with CMake can be found in §10.

When SUNDIALS is built with profiling enabled and **without Caliper**, then the environment variable `SUNPROFILER_PRINT` can be utilized to enable/disable the printing of profiler information. Setting `SUNPROFILER_PRINT=1` will cause the profiling information to be printed to stdout when the SUNDIALS simulation context is freed. Setting `SUNPROFILER_PRINT=0` will result in no profiling information being printed unless the *SUNProfiler_Print()* function is called explicitly. By default, `SUNPROFILER_PRINT` is assumed to be `0`. `SUNPROFILER_PRINT` can also be set to a file path where the output should be printed.

If Caliper is enabled, then users should refer to the Caliper documentation for information on getting profiler output. In most cases, this involves setting the `CALI_CONFIG` environment variable.

Warning: While the SUNDIALS profiling scheme is relatively lightweight, enabling profiling can still negatively impact performance. As such, it is recommended that profiling is enabled judiciously.

4.2.2 Profiler API

The primary way of interacting with the SUNDIALS profiler is through the following macros:

```
SUNDIALS_MARK_FUNCTION_BEGIN(profobj)
SUNDIALS_MARK_FUNCTION_END(profobj)
SUNDIALS_WRAP_STATEMENT(profobj, name, stmt)
SUNDIALS_MARK_BEGIN(profobj, name)
SUNDIALS_MARK_END(profobj, name)
```

Additionally, in C++ applications, the follow macro is available:

```
SUNDIALS_CXX_MARK_FUNCTION(profobj)
```

These macros can be used to time specific functions or code regions. When using the `*_BEGIN` macros, it is important that a matching `*_END` macro is placed at all exit points for the scope/function. The `SUNDIALS_CXX_MARK_FUNCTION` macro only needs to be placed at the beginning of a function, and leverages RAII to implicitly end the region.

The `profobj` argument to the macro should be a `SUNProfiler` object, i.e. an instance of the struct

typedef struct _SUNProfiler *__SUNProfiler__

When SUNDIALS is built with profiling, a default profiling object is stored in the SUNContext object and can be accessed with a call to *SUNContext_GetProfiler()*.

The name argument should be a unique string indicating the name of the region/function. It is important that the name given to the *_BEGIN macros matches the name given to the *_END macros.

In addition to the macros, the following methods of the SUNProfiler class are available.

int **SUNProfiler_Create**(void *comm, const char *title, *SUNProfiler* *p)
Creates a new SUNProfiler object.

Arguments:

- comm – a pointer to the MPI communicator if MPI is enabled, otherwise can be NULL
- title – a title or description of the profiler
- p – [in,out] On input this is a pointer to a SUNProfiler, on output it will point to a new SUNProfiler instance

Returns:

- Returns zero if successful, or non-zero if an error occurred

int **SUNProfiler_Free**(*SUNProfiler* *p)
Frees a SUNProfiler object.

Arguments:

- p – [in,out] On input this is a pointer to a SUNProfiler, on output it will be NULL

Returns:

- Returns zero if successful, or non-zero if an error occurred

int **SUNProfiler_Begin**(*SUNProfiler* p, const char *name)
Starts timing the region indicated by the name.

Arguments:

- p – a SUNProfiler object
- name – a name for the profiling region

Returns:

- Returns zero if successful, or non-zero if an error occurred

int **SUNProfiler_End**(*SUNProfiler* p, const char *name)
Ends the timing of a region indicated by the name.

Arguments:

- p – a SUNProfiler object
- name – a name for the profiling region

Returns:

- Returns zero if successful, or non-zero if an error occurred

int **SUNProfiler_Print**(*SUNProfiler* p, FILE *fp)
Prints out a profiling summary. When constructed with an MPI comm the summary will include the average and maximum time per rank (in seconds) spent in each marked up region.

Arguments:

- p – a SUNProfiler object

- `fp` – the file handler to print to

Returns:

- Returns zero if successful, or non-zero if an error occurred

4.2.3 Example Usage

The following is an excerpt from the CVODE example code `examples/cvode/serial/cvAdvDiff_bnd.c`. It is applicable to any of the SUNDIALS solver packages.

```
SUNContext ctx;
SUNProfiler profobj;

/* Create the SUNDIALS context */
retval = SUNContext_Create(NULL, &ctx);

/* Get a reference to the profiler */
retval = SUNContext_GetProfiler(ctx, &profobj);

/* ... */

SUNDIALS_MARK_BEGIN(profobj, "Integration loop");
umax = N_VMaxNorm(u);
PrintHeader(reltol, abstol, umax);
for(iout=1, tout=T1; iout <= NOUT; iout++, tout += DTOUT) {
   retval = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
   umax = N_VMaxNorm(u);
   retval = CVodeGetNumSteps(cvode_mem, &nst);
   PrintOutput(t, umax, nst);
}
SUNDIALS_MARK_END(profobj, "Integration loop");
PrintFinalStats(cvode_mem);  /* Print some final statistics   */
```

4.2.4 Other Considerations

If many regions are being timed, it may be necessary to increase the maximum number of profiler entries (the default is `2560`). This can be done by setting the environment variable `SUNPROFILER_MAX_ENTRIES`.

4.3 SUNDIALS version information

SUNDIALS provides additional utilities to all packages, that may be used to retrieve SUNDIALS version information at runtime.

int **SUNDIALSGetVersion**(char *version, int len)
This routine fills a string with SUNDIALS version information.

Arguments:

- *version* – character array to hold the SUNDIALS version information.
- *len* – allocated length of the *version* character array.

Return value:

- 0 if successful
- -1 if the input string is too short to store the SUNDIALS version

Notes: An array of 25 characters should be sufficient to hold the version information.

int **SUNDIALSGetVersionNumber**(int *major, int *minor, int *patch, char *label, int len)
This routine sets integers for the SUNDIALS major, minor, and patch release numbers and fills a string with the release label if applicable.

Arguments:

- *major* – SUNDIALS release major version number.
- *minor* – SUNDIALS release minor version number.
- *patch* – SUNDIALS release patch version number.
- *label* – string to hold the SUNDIALS release label.
- *len* – allocated length of the *label* character array.

Return value:

- 0 if successful
- -1 if the input string is too short to store the SUNDIALS label

Notes: An array 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*.

4.4 SUNDIALS Fortran Interface

SUNDIALS provides modern, Fortran 2003 based, interfaces as Fortran modules to most of the C API including:

- All of the time-stepping modules in ARKODE:
  - The `farkode_arkstep_mod`, `farkode_erkstep_mod`, and `farkode_mristep_mod` modules provide interfaces to the ARKStep, ERKStep, and MRIStep integrators respectively.
  - The `farkode_mod` module interfaces to the components of ARKODE which are shared by the time-stepping modules.
- CVODE via the `fcvode_mod` module.
- CVODES via the `fcvodes_mod` module.
- IDA via the `fida_mod` module.
- IDAS via the `fidas_mod` module.
- KINSOL via the `fkinsol_mod` module.

Additionally, all of the SUNDIALS base classes (*N_Vector*, *SUNMatrix*, and *SUNLinearSolver*) include Fortran interface modules. A complete list of class implementations with Fortran 2003 interface modules is given in Table 4.1.

An interface module can be accessed with the `use` statement, e.g.

```
use fcvode_mod
use fnvector_openmp_mod
```

and by linking to the Fortran 2003 library in addition to the C library, e.g. `libsundials_fnvecpenmp_mod.<so|a>`, `libsundials_nvecopenmp.<so|a>`, `libsundials_fcvode_mod.<so|a>` and `libsundials_cvode.<so|a>`.

The Fortran 2003 interfaces leverage the `iso_c_binding` module and the `bind(C)` attribute to closely follow the SUNDIALS C API (modulo language differences). The SUNDIALS classes, 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()* or `FCVodeCreate` instead of `CVodeCreate`. 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 §4.4.2. A discussion on the topic of equivalent data types in C and Fortran 2003 is presented in §4.4.1.

Further information on the Fortran 2003 interfaces specific to the *N_Vector*, *SUNMatrix*, and *SUNLinearSolver* classes is given alongside the C documentation (§6, §7, and §8, respectively). For details on where the Fortran 2003 module (`.mod`) files and libraries are installed see §10.

The Fortran 2003 interface modules were generated with SWIG Fortran [22], a fork of SWIG. Users who are interested in the SWIG code used in the generation process should contact the SUNDIALS development team.

Table 4.1: List of SUNDIALS Fortran 2003 interface modules

Class/Module	Fortran 2003 Module Name
ARKODE	`farkode_mod`
ARKODE::ARKSTEP	`farkode_arkstep_mod`
ARKODE::ERKSTEP	`farkode_erkstep_mod`
ARKODE::MRISTEP	`farkode_mristep_mod`
CVODE	`fcvode_mod`
CVODES	`fcvodes_mod`
IDA	`fida_mod`
IDAS	`fidas_mod`
KINSOL	`fkinsol_mod`
NVECTOR	`fsundials_nvector_mod`
NVECTOR_SERIAL	`fnvector_serial_mod`
NVECTOR_OPENMP	`fnvector_openmp_mod`
NVECTOR_PTHREADS	`fnvector_pthreads_mod`
NVECTOR_PARALLEL	`fnvector_parallel_mod`
NVECTOR_PARHYP	Not interfaced
NVECTOR_PETSC	Not interfaced
NVECTOR_CUDA	Not interfaced
NVECTOR_RAJA	Not interfaced
NVECTOR_SYCL	Not interfaced
NVECTOR_MANVECTOR	`fnvector_manyvector_mod`
NVECTOR_MPIMANVECTOR	`fnvector_mpimanyvector_mod`
NVECTOR_MPIPLUSX	`fnvector_mpiplusx_mod`
SUNMATRIX	`fsundials_matrix_mod`
SUNMATRIX_BAND	`fsunmatrix_band_mod`
SUNMATRIX_DENSE	`fsunmatrix_dense_mod`
SUNMATRIX_MAGMADENSE	Not interfaced
SUNMATRIX_ONEMKLDENSE	Not interfaced
SUNMATRIX_SPARSE	`fsunmatrix_sparse_mod`
SUNLINSOL	`fsundials_linearsolver_mod`
SUNLINSOL_BAND	`fsunlinsol_band_mod`
SUNLINSOL_DENSE	`fsunlinsol_dense_mod`
SUNLINSOL_LAPACKBAND	Not interfaced
SUNLINSOL_LAPACKDENSE	Not interfaced
SUNLINSOL_MAGMADENSE	Not interfaced
SUNLINSOL_ONEMKLDENSE	Not interfaced

continues on next page

Table 4.1 – continued from previous page

Class/Module	Fortran 2003 Module Name
SUNLINSOL_KLU	fsunlinsol_klu_mod
SUNLINSOL_SLUMT	Not interfaced
SUNLINSOL_SLUDIST	Not interfaced
SUNLINSOL_SPGMR	fsunlinsol_spgmr_mod
SUNLINSOL_SPFGMR	fsunlinsol_spfgmr_mod
SUNLINSOL_SPBCGS	fsunlinsol_spbcgs_mod
SUNLINSOL_SPTFQMR	fsunlinsol_sptfqmr_mod
SUNLINSOL_PCG	fsunlinsol_pcg_mof
SUNNONLINSOL	fsundials_nonlinearsolver_mod
SUNNONLINSOL_NEWTON	fsunnonlinsol_newton_mod
SUNNONLINSOL_FIXEDPOINT	fsunnonlinsol_fixedpoint_mod
SUNNONLINSOL_PETSCSNES	Not interfaced

4.4.1 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 classes 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 4.2 presents a summary of the type equivalencies with the parameter direction in mind.

Warning: Currently, the Fortran 2003 interfaces are only compatible with SUNDIALS builds where the `realtype` is double-precision the `sunindextype` size is 64-bits.

Table 4.2: 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)
realtype	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	real(c_int), dimension(*)
int*	return	real(c_int), pointer, dimension(:)
long*	in, inout, out	real(c_long), dimension(*)
long*	return	real(c_long), pointer, dimension(:)
realtype*	in, inout, out	real(c_double), dimension(*)
realtype*	return	real(c_double), pointer, dimension(:)
sunindextype*	in, inout, out	real(c_long), dimension(*)
sunindextype*	return	real(c_long), pointer, dimension(:)
realtype[]	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)

continues on next page

Table 4.2 – continued from previous page

C Type	Parameter Direction	Fortran 2003 type
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)

4.4.2 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, §4.4.1 discusses equivalencies of data types in the two languages.

4.4.2.1 Creating generic SUNDIALS objects

In the C API a SUNDIALS class, 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. For example, 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, sunctx);
```

Fortran code:

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

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

4.4.2.2 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;
```

(continues on next page)

(continued from previous page)

```
/* create a new serial vector */
x = N_VNew_Serial(N, sunctx);

/* 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)

! create a new serial vector
x => FN_VNew_Serial(x, sunctx)

! 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)
```

4.4.2.3 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 *cvode_mem;

ierr = CVodeSetUserData(cvode_mem, udata);
```

Fortran code:

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

(continues on next page)

(continued from previous page)

```
ierr = FARKStepSetUserData(arkode_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.

4.4.2.4 Passing `NULL` to optional parameters

In the SUNDIALS C API some functions have optional parameters that a caller can pass as `NULL`. If the optional parameter is of a type that is equivalent to a Fortran `type(c_ptr)` (see §4.4.1), 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

! Disassociate A
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)
```

4.4.2.5 Working with `N_Vector` arrays

Arrays of *N_Vector* objects are interfaced to Fortran 2003 as an 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 (CVODES and IDAS). 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;

/* Create an array of N_Vectors */
vecs = N_VCloneVectorArray(count, x);

/* Fill each array with ones */
```

(continues on next page)

(continued from previous page)

```
for (int i = 0; i < count; ++i)
  N_VConst(vecs[i], 1.0);
```

Fortran code:

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

! Create an array of N_Vectors
vecs = FN_VCloneVectorArray(count, x)

! Fill each array with ones
do index = 0,count-1
  xi => FN_VGetVecAtIndexVectorArray(vecs, index)
  call FN_VConst(xi, 1.d0)
enddo
```

SUNDIALS also provides the functions *N_VSetVecAtIndexVectorArray()* and *N_VNewVectorArray()* for working with `N_Vector` arrays, that have corresponding Fortran interfaces `FN_VSetVecAtIndexVectorArray` and `FN_VNewVectorArray`, respectively. 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 *N_VGetVecAtIndexVectorArray()* (wrapped as `FN_VGetVecAtIndexVectorArray`) are further described in §6.1.1.

4.4.2.6 Providing file pointers

There are a few functions in the SUNDIALS C API which 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`.

FILE ***SUNDIALSFileOpen**(filename, mode)
The function allocates a `FILE*` by calling the C function `fopen` with the provided filename and I/O mode.

Arguments:

- `filename` – the full path to the file, that should have Fortran type `character(kind=C_CHAR, len=*)`.
- `mode` – the I/O mode to use for the file. This should have the Fortran type `character(kind=C_CHAR, len=*)`. The string begins with one of the following characters:
  - `r` to open a text file for reading
  - `r+` to open a text file for reading/writing
  - `w` to truncate a text file to zero length or create it for writing
  - `w+` to open a text file for reading/writing or create it if it does not exist
  - `a` to open a text file for appending, see documentation of `fopen` for your system/compiler
  - `a+` to open a text file for reading/appending, see documentation for `fopen` for your system/compiler

Return value:

- The function returns a `type(C_PTR)` which holds a C `FILE*`.

void **SUNDIALSFileClose**(fp)
The function deallocates a C `FILE*` by calling the C function `fclose` with the provided pointer.

Arguments:

- `fp` – the C `FILE*` that was previously obtained from `fopen`. This should have the Fortran type `type(c_ptr)`.

4.4.3 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 that was used to generate the modules.

4.5 Features for GPU Accelerated Computing

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 Chapters §6, §7, and §8) 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.

4.5.1 SUNDIALS GPU Programming Model

As described in [2], 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 suffer. The exception to this rule is if some form of hybrid data partitioning (achievable with the NVECTOR_MANYVECTOR, see §6.16) is utilized.

SUNDIALS provides many native shared features and modules that are GPU-enabled. Currently, these include the NVIDIA CUDA platform [38], AMD ROCm/HIP [35], and Intel oneAPI [36]. Table 4.3–Table 4.5 summarize 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*, or *SUNLinearSolver* implementation, and the capabilties will be leveraged since SUNDIALS operates on data through these APIs.

In addition, SUNDIALS provides a memory management helper module (see §9) to support applications which implement their own memory management or memory pooling.

Table 4.3: List of SUNDIALS GPU-enabled `N_Vector` Modules

Module	CUDA	ROCm/HIP	oneAPI	Unmanaged Memory	UVM
NVECTOR_CUDA	X			X	X
NVECTOR_HIP	X	X		X	X
NVECTOR_RAJA	X	X	X	X	X
NVECTOR_SYCL	X[3]	X[3]	X	X	X
NVECTOR_OPENMPDEV	X	X[2]	X[2]	X	

Table 4.4: List of SUNDIALS GPU-enabled `SUNMatrix` Modules

Module	CUDA	ROCm/HIP	oneAPI	Unmanaged Memory	UVM
SUNMATRIX_CUSPARSE	X			X	X
SUNMATRIX_MAGMADENSE	X	X		X	X
SUNMATRIX_ONEMKLDENSE	X[3]	X[3]	X	X	X

Table 4.5: List of SUNDIALS GPU-enabled `SUNLinearSolver` Modules

Module	CUDA	ROCm/HIP	oneAPI	Unmanaged Memory	UVM
SUNLINSOL_CUSOLVERSP	X			X	X
SUNLINSOL_MAGMADENSE	X			X	X
SUNLINSOL_ONEMKLDENSE	X[3]	X[3]	X	X	X
SUNLINSOL_SPGMR	X[1]	X[1]	X[1]	X[1]	X[1]
SUNLINSOL_SPFGMR	X[1]	X[1]	X[1]	X[1]	X[1]
SUNLINSOL_SPTFQMR	X[1]	X[1]	X[1]	X[1]	X[1]
SUNLINSOL_SPBCGS	X[1]	X[1]	X[1]	X[1]	X[1]
SUNLINSOL_PCG	X[1]	X[1]	X[1]	X[1]	X[1]

Notes regarding the above tables:

1. This module inherits support from the NVECTOR module used
2. Support for ROCm/HIP and oneAPI are currently untested.
3. Support for CUDA and ROCm/HIP are currently untested.

In addition, note that implicit UVM (i.e. `malloc` returning UVM) is not accounted for.

4.5.2 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 `N_Vector` 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 `SUNLinearSolver` linear solver (if applicable).
3. Utilize a GPU-enabled `SUNMatrix` implementation (if using a matrix-based linear solver).
4. Utilize a GPU-enabled `SUNNonlinearSolver` nonlinear solver (if applicable).
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 evalution function, or the preconditioner evaulation 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 *N_Vector* object that is passed from SUNDIALS to the user-supplied function.

Users should refer to the above tables for a complete list of GPU-enabled native SUNDIALS modules.

Chapter 5

Using KINSOL for the Solution of Nonlinear Systems

This section is concerned with the use of KINSOL for the solution of nonlinear systems.

The following sections treat the header files and the layout of the user's main program, and provide descriptions of the KINSOL user-callable functions and user-supplied functions. The sample programs described in the companion document [8] may also be helpful. Those codes may be used as templates (with the removal of some lines used in testing) and are included in the KINSOL package.

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 §11.

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

5.1 Access to library and header files

At this point, it is assumed that the installation of KINSOL, following the procedure described in §10, 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.<so|a>
<libdir>/libsundials_nvec*.<so|a>
<libdir>/libsundials_sunmat*.<so|a>
<libdir>/libsundials_sunlinsol*.<so|a>
<libdir>/libsundials_sunnonlinsol*.<so|a>
```

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

```
<incdir>/kinsol
<incdir>/sundials
<incdir>/nvector
<incdir>/sunmatrix
<incdir>/sunlinsol
<incdir>/sunnonlinsol
```

The directories `libdir` and `incdir` are the install library and include directories, respectively. For a default installation, these are `<instdir>/lib` or `<instdir>/lib64` and `<instdir>/include`, respectively, where `instdir` is the directory where SUNDIALS was installed (see §10).

5.2 Data Types

The header file `sundials_types.h` contains the definition of the types:

- *realtype* – the floating-point type used by the SUNDIALS packages
- *sunindextype* – the integer type used for vector and matrix indices
- *booleantype* – the type used for logic operations within SUNDIALS

5.2.1 Floating point types

type **realtype**

The type `realtype` can be `float`, `double`, or `long double`, with the default being `double`. The user can change the precision of the arithmetic used in the SUNDIALS solvers at the configuration stage (see *SUNDIALS_PRECISION*).

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 of `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 C or C++ code can use SUNDIALS without modifying the code to use `realtype`, `RCONST`, or the `SUNR` macros so long as the SUNDIALS libraries are built to use the corresponding precision (see §10.1.2).

5.2.2 Integer types used for indexing

type **sunindextype**

The type `sunindextype` is used for indexing array entries in SUNDIALS modules as well as for storing the total problem size (*e.g.*, vector lengths and matrix sizes). During configuration `sunindextype` may be selected to be either a 32- or 64-bit *signed* integer with the default being 64-bit (see *SUNDIALS_INDEX_SIZE*).

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` (see *SUNDIALS_INDEX_TYPE*).

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 C or 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 §10.1.2).

5.2.3 Boolean type

type **booleantype**

As ANSI C89 (ISO C90) does not have a built-in boolean data type, SUNDIALS defines the type `booleantype` as an `int`.

The advantage of using the name booleantype (instead of int) is an increase in code readability. It also allows the programmer to make a distinction between int and boolean data. Variables of type `booleantype` are intended to have only the two values `SUNFALSE` (`0`) and `SUNTRUE` (`1`).

5.3 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 main header file for kinsol, which defines the types and various constants, and includes function prototypes. This includes the header file for KINLS, `kinsol/kinsol_ls.h`.

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

The calling program must also include an `N_Vector` implementation header file, of the form `nvector/nvector_*.h` (see §6 for more information). This file in turn includes the header file `sundials_nvector.h` which defines the abstract 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. If the linear solver is matrix-based, the linear solver header will also include a header file of the from `sunmatrix/sunmatrix_*.h` where `*` is the name of the matrix implementation compatible with the linear solver. The matrix header file provides access to the relevant matrix functions/macros and in turn includes the header file `sundials_matrix.h` which defines the abstract matrix data type.

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

5.4 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 `N_Vector`, `SUNMatrix`, and `SUNLinearSolver` implementations used. For the steps that are not, refer to §6, §7, and §8 for the specific name of the function to be called or macro to be referenced.

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

   For example, call `MPI_Init` to initialize MPI if used.

2. **Create the SUNDIALS context object**

   Call *SUNContext_Create()* to allocate the `SUNContext` object.

3. **Set the problem dimensions etc.**

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

4. **Create the vector with the initial guess**

   Construct an `N_Vector` of initial guess values using the appropriate functions defined by the particular `N_Vector` implementation (see §6 for details).

   For native SUNDIALS vector implementations, use a call of the form `y0 = N_VMake_***(..., ydata)` if the array containing the initial values of y already exists. Otherwise, create a new vector by making a call of the form `N_VNew_***(...)`, and then set its elements by accessing the underlying data with a call of the form `ydata = N_VGetArrayPointer(y0)`. Here, `***` is the name of the vector implementation.

   For *hypre*, PETSc, and Trilinos vector wrappers, first create and initialize the underlying vector, and then create an `N_Vector` wrapper with a call of the form `y0 = N_VMake_***(yvec)`, where `yvec` is a *hypre*, PETSc, or Trilinos vector. Note that calls like `N_VNew_***(...)` and `N_VGetArrayPointer(...)` are not available for these vector wrappers.

5. **Create matrix object** (*if appropriate*)

   If a linear solver is required (e.g., when using the default Newton solver) and the linear solver will be a matrix-based linear solver, then a template Jacobian matrix must be created by calling the appropriate constructor defined by the particular `SUNMatrix` implementation.

   For the native SUNDIALS `SUNMatrix` implementations, the matrix object may be created using a call of the form `SUN***Matrix(...)` where `***` is the name of the matrix (see §7 for details).

6. **Create linear solver object** (*if appropriate*)

   If a linear solver is required (e.g., when using the default Newton solver), then the desired linear solver object must be created by calling the appropriate constructor defined by the particular `SUNLinearSolver` implementation.

   For any of the native SUNDIALS `SUNLinearSolver` implementations, the linear solver object may be created using a call of the form `SUNLinearSolver LS = SUNLinSol_***(...);` where `***` is the name of the linear solver (see §8 for details).

7. **Create KINSOL object**

   Call *KINCreate()* to create the KINSOL solver object.

8. **Initialize KINSOL solver**

   Call *KINInit()* to allocate internal memory.

9. **Attach the linear solver** (*if appropriate*)

   If a linear solver was created above, initialize the KINLS linear solver interface by attaching the linear solver object (and matrix object, if applicable) with *KINSetLinearSolver()*.

10. **Set linear solver optional inputs** (*if appropriate*)

    See Table 5.1 for KINLS optional inputs and Chapter §8 for linear solver specific optional inputs.

11. **Set optional inputs**

    Call `KINSet***` functions to change any optional inputs that control the behavior of KINSOL from their default values. See §5.5.4 for details.

12. **Solve problem**

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

    See *KINSol()* for details.

13. **Get optional outputs**

    Call `KINGet***` functions to obtain optional output. See §5.5.5 for details.

14. **Deallocate memory**

    Upon completion of the integration call the following, as necessary, to free any objects or memory allocated above:

    - Call *N_VDestroy()* to free vector objects.
    - Call *SUNMatDestroy()* to free matrix objects.
    - Call *SUNLinSolFree()* to free linear solvers objects.
    - Call `SUNNonlinSolFree()` to free nonlinear solvers objects.
    - Call *KINFree()* to free the memory allocated by KINSOL.
    - Call *SUNContext_Free()* to free the `SUNContext` object

15. **Finalize MPI, if used**

    Call `MPI_Finalize` to terminate MPI.

5.5 User-callable functions

This section describes the KINSOL functions that are called by the user to setup and then solve an IVP. Some of these are required. However, starting with §5.5.4, the functions listed involve optional inputs/outputs or restarting, and those paragraphs may be skipped for a casual use of KINSOL. In any case, refer to §5.4 for the correct order of these calls.

On an error, each user-callable function returns a negative value and sends an error message to the error handler routine, which prints the message on `stderr` by default. However, the user can set a file as error output or can provide his own error handler function (see §5.5.4).

5.5.1 KINSOL initialization and deallocation functions

void **KINCreate**(*SUNContext* sunctx)

The function *KINCreate()* instantiates a KINSOL solver object.

Arguments:

- sunctx – the *SUNContext* object (see §4.1)

Return value:

- void

int **KINInit**(void *kin_mem, *KINSysFn* func, *N_Vector* tmpl)

The function *KINInit()* specifies the problem-defining function, allocates internal memory, and initializes KINSOL.

Arguments:

- kin_mem – pointer to the KINSOL memory block returned by *KINCreate()*.
- func – is the CC function which computes the system function $F(u)$ (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 §5.6.1).
- tmpl – 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:

- 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.

void **KINFree**(void **kin_mem)

The function *KINFree()* frees the pointer allocated by a previous call to *KINCreate()*.

Arguments:

- kin_mem – pointer to the KINSOL solver object.

Return value:

- void

5.5.2 Linear solver specification functions

As previously explained, Newton and Picard iterations require the solution of linear systems of the form $J\delta = -F$. Solution of these linear systems is handled using the KINLS linear solver interface. This interface supports all valid SUNLinearSolver modules. Here, matrix-based SUNLinearSolver modules utilize SUNMatrix objects to store the Jacobian matrix $J = F'(u)$ and factorizations used throughout the solution process. Conversely, matrix-free SUNLinearSolver 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 §5.5.4 and §5.6. 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 SUNLinearSolver object and call the function *KINSetLinearSolver()*, as documented below. To create the SUNLinearSolver object, the user may call one of the SUNDIALS-packaged SUNLinearSolver module constructor routines via a call of the form

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

For a current list of such constructor routines see §8.

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 SUNLinearSolver module in question, as described in Chapters §7 and §8.

Once this solver object has been constructed, the user should attach it to KINSOL via a call to *KINSetLinearSolver()*. The first argument passed to this function is the KINSOL memory pointer returned by *KINCreate()*; the second argument is the desired SUNLinearSolver object to use for solving Newton or Picard systems. The third argument is an optional SUNMatrix object to accompany matrix-based SUNLinearSolver 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.

int **KINSetLinearSolver**(void *kin_mem, *SUNLinearSolver* LS, *SUNMatrix* J)

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

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- LS – SUNLINSOL object to use for solving Newton linear systems.
- J – SUNMATRIX object for used as a template for the Jacobian (or NULL if not applicable).

Return value:

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

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

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

5.5.3 KINSOL solver function

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

int **KINSol**(void *kin_mem, *N_Vector* u, int strategy, *N_Vector* u_scale, *N_Vector* f_scale)

The function *KINSol()* computes an approximate solution to the nonlinear system.

Arguments:

- `kin_mem` – pointer to the KINSOL memory block.
- `u` – 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` – 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` – vector containing diagonal elements of scaling matrix D_u for vector `u` chosen so that the components of $D_u\ u$ (as a matrix multiplication) all have roughly the same magnitude when `u` is close to a root of $F(u)$.
- `f_scale` – vector containing diagonal elements of scaling matrix D_F for $F(u)$ chosen so that the components of $D_F\ F(u)$ (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:

- `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*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 *KINSetScaledStepTol()* in §5.5.4 to change `scsteptol` from its default value).
- `KIN_MEM_NULL` – The KINSOL memory block pointer was `NULL`.
- `KIN_ILL_INPUT` – An input parameter was invalid.
- `KIN_NO_MALLOC` – The KINSOL memory was not allocated by a call to *KINCreate()*.
- `KIN_MEM_FAIL` – A memory allocation failed.
- `KIN_LINESEARCH_NONCONV` – The line search algorithm was unable to find an iterate sufficiently distinct from the current iterate, or could not find an iterate satisfying the sufficient decrease condition. Failure to satisfy the sufficient decrease condition could mean the current iterate is "close" to an approximate solution of the given nonlinear system, the difference approximation of the matrix-vector product $J(u)\ v$ is inaccurate, or the real scalar `scsteptol` is too large.
- `KIN_MAXITER_REACHED` – The maximum number of nonlinear iterations has been reached.
- `KIN_MXNEWT_5X_EXCEEDED` – Five consecutive steps have been taken that satisfy the inequality $\|D_u p\|_{L2} > 0.99$ `mxnewtstep` , where p denotes the current step and `mxnewtstep` is a scalar upper bound on the scaled step length. Such a failure may mean that $\|D_F F(u)\|_{L2}$ asymptotes from above to a positive value, or the real scalar `mxnewtstep` is too small.

- KIN_LINESEARCH_BCFAIL – The line search algorithm was unable to satisfy the "beta-condition" for MXNBCF+1 nonlinear iterations (not necessarily consecutive), which may indicate the algorithm is making poor progress.
- KIN_LINSOLV_NO_RECOVERY – The user-supplied routine psolve encountered a recoverable error, but the preconditioner is already current.
- KIN_LINIT_FAIL – The 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 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.

5.5.4 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 5.1 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.

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 retval < 0 will catch any error.

Table 5.1: 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

continues on next page

Table 5.1 – continued from previous page

Optional input	Function name	Default
Lower bound on ϵ	*KINSetNoMinEps()*	SUNFALSE
Max. scaled length of Newton step	*KINSetMaxNewtonStep()*	$1000\|D_u u_0\|_2$
Max. number of β-condition failures	*KINSetMaxBetaFails()*	10
Rel. error for D.Q. Jv	*KINSetRelErrFunc()*	$\sqrt{\text{uround}}$
Function-norm stopping tolerance	*KINSetFuncNormTol()*	$\text{uround}^{1/3}$
Scaled-step stopping tolerance	*KINSetScaledStepTol()*	$\text{uround}^{2/3}$
Inequality constraints on solution	*KINSetConstraints()*	NULL
Nonlinear system function	*KINSetSysFunc()*	none
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
Anderson Acceleration orthogonalization routine	*KINSetOrthAA()*	KIN_ORTH_MGS
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

int **KINSetErrFile**(void *kin_mem, FILE *errfp)

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 – pointer to the KINSOL memory block.
- errfp – pointer to output file.

Return value:

- 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.

Warning: 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.

int **KINSetErrHandlerFn**(void *kin_mem, *KINErrHandlerFn* ehfun, void *eh_data)

The function *KINSetErrHandlerFn()* specifies the optional user-defined function to be used in handling error messages.

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- ehfun – is the user's CC error handler function (see §5.6.2).
- eh_data – pointer to user data passed to ehfun every time it is called.

Return value:

- 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.

int **KINSetInfoFile**(void *kin_mem, FILE *infofp)

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

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- infofp – pointer to output file.

Return value:

- 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.

int **KINSetInfoHandlerFn**(void *kin_mem, *KINInfoHandlerFn* ihfun, void *ih_data)

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

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- ihfun – is the user's CC information handler function (see §5.6.3).
- ih_data – pointer to user data passed to ihfun every time it is called.

Return value:

- 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).

int **KINSetPrintLevel**(void *kin_mem, int printfl)

The function *KINSetPrintLevel()* specifies the level of verbosity of the output.

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- printfl – flag indicating the level of verbosity. Must be one of:

  0 – no information is 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:

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

Notes: The default value for printfl is 0.

int **KINSetUserData**(void *kin_mem, void *user_data)

The function *KINSetUserData()* specifies the pointer to user-defined memory that is to be passed to all user-supplied functions.

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- user_data – pointer to the user-defined memory.

Return value:

- 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.

> **Warning:** 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.

int **KINSetNumMaxIters**(void *kin_mem, long int mxiter)

The function *KINSetNumMaxIters()* specifies the maximum number of nonlinear iterations allowed.

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- mxiter – maximum number of nonlinear iterations.

Return value:

- 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 $= 200$.

int **KINSetNoInitSetup**(void *kin_mem, *booleantype* noInitSetup)

The function *KINSetNoInitSetup()* specifies whether an initial call to the preconditioner or Jacobian setup function should be made or not.

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- noInitSetup – flag controlling whether an initial call to the preconditioner or Jacobian setup function is made (pass SUNFALSE) or not made (pass SUNTRUE).

Return value:

- 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.

int **KINSetNoResMon**(void *kin_mem, *booleantype* noNNIResMon)

The function *KINSetNoResMon()* specifies whether or not the nonlinear residual monitoring scheme is used to control Jacobian updating

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- noNNIResMon – flag controlling whether residual monitoring is used (pass SUNFALSE) or not used (pass SUNTRUE).

Return value:

- 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.

Warning: Residual monitoring is only available for use with matrix-based linear solver modules.

int **KINSetMaxSetupCalls**(void *kin_mem, long int msbset)

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 – pointer to the KINSOL memory block.
- msbset – maximum number of nonlinear iterations without a call to the preconditioner or Jacobian setup function. Pass 0 to indicate the default.

Return value:

- 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()*).

int **KINSetMaxSubSetupCalls**(void *kin_mem, long int msbsetsub)

The function *KINSetMaxSubSetupCalls()* specifies the maximum number of nonlinear iterations between checks by the residual monitoring algorithm.

Arguments:

- `kin_mem` – pointer to the KINSOL memory block.
- `msbsetsub` – maximum number of nonlinear iterations without checking the nonlinear residual. Pass 0 to indicate the default.

Return value:

- `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`.

Warning: Residual monitoring is only available for use with matrix-based linear solver modules.

int **KINSetEtaForm**(void *kin_mem, int etachoice)

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` – pointer to the KINSOL memory block.
- `etachoice` – 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:

- `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 $\eta_{\text{min}} \leq \eta_n \leq \eta_{\text{max}}$:

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

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

int **KINSetEtaConstValue**(void *kin_mem, *realtype* eta)

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

Arguments:

- `kin_mem` – pointer to the KINSOL memory block.
- `eta` – constant value for η. Pass 0.0 to indicate the default.

Return value:

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

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

int **KINSetEtaParams**(void *kin_mem, *realtype* egamma, *realtype* ealpha)

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

Arguments:

- `kin_mem` – pointer to the KINSOL memory block.
- `egamma` – value of the γ parameter. Pass 0.0 to indicate the default.
- `ealpha` – value of the α parameter. Pass 0.0 to indicate the default.

Return value:

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

Notes: The default values for `egamma` and `ealpha` are 0.9 and 2.0, respectively. The legal values are $0.0 <$ `egamma` ≤ 1.0 and $1.0 <$ `ealpha` ≤ 2.0.

int **KINSetResMonConstValue**(void *kin_mem, *realtype* omegaconst)

The function *KINSetResMonConstValue()* specifies the constant value for ω when using residual monitoring.

Arguments:

- `kin_mem` – pointer to the KINSOL memory block.
- `omegaconst` – constant value for ω. Passing 0.0 results in using Eqn. (2.4).

Return value:

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

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

int **KINSetResMonParams**(void *kin_mem, *realtype* omegamin, *realtype* omegamax)

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

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- omegamin – value of the ω_{min} parameter. Pass 0.0 to indicate the default.
- omegamax – value of the ω_{max} parameter. Pass 0.0 to indicate the default.

Return value:

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

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

int **KINSetNoMinEps**(void *kin_mem, *booleantype* noMinEps)

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 – pointer to the KINSOL memory block.
- noMinEps – 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:

- 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, is applied to ϵ (see *KINSetFuncNormTol()* below).

int **KINSetMaxNewtonStep**(void *kin_mem, *realtype* mxnewtstep)

The function *KINSetMaxNewtonStep()* specifies the maximum allowable scaled length of the Newton step.

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- mxnewtstep – maximum scaled step length (≥ 0.0). Pass 0.0 to indicate the default.

Return value:

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

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

int **KINSetMaxBetaFails**(void *kin_mem, *realtype* mxnbcf)

The function *KINSetMaxBetaFails()* specifies the maximum number of β-condition failures in the linesearch algorithm.

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- mxnbcf – maximum number of β -condition failures. Pass 0.0 to indicate the default.

Return value:

- 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 $= 10$.

int **KINSetRelErrFunc**(void *kin_mem, *realtype* relfunc)

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. (2.6)] or the Jacobian-vector product [see Eq. (2.8)]. The value stored is $\sqrt{\texttt{relfunc}}$.

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- relfunc – relative error in $F(u)$ (relfunc ≥ 0.0). Pass 0.0 to indicate the default.

Return value:

- KIN_SUCCESS – The optional value has been successfully set.
- KIN_MEM_NULL – The kin_mem pointer is NULL.
- KIN_ILL_INPUT – The relative error was negative.

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

int **KINSetFuncNormTol**(void *kin_mem, *realtype* fnormtol)

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 – pointer to the KINSOL memory block.
- fnormtol – tolerance for stopping based on scaled function norm (≥ 0.0). Pass 0.0 to indicate the default.

Return value:

- KIN_SUCCESS – The optional value has been successfully set.
- KIN_MEM_NULL – The kin_mem pointer is NULL.
- KIN_ILL_INPUT – The tolerance was negative.

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

int **KINSetScaledStepTol**(void *kin_mem, *realtype* scsteptol)

The function *KINSetScaledStepTol()* specifies the scalar used as a stopping tolerance on the minimum scaled step length.

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- scsteptol – tolerance for stopping based on scaled step length (≥ 0.0). Pass 0.0 to indicate the default.

Return value:

- KIN_SUCCESS – The optional value has been successfully set.
- KIN_MEM_NULL – The kin_mem pointer is NULL.
- KIN_ILL_INPUT – The tolerance was non-positive.

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

int **KINSetConstraints**(void *kin_mem, *N_Vector* constraints)

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

Arguments:

- `kin_mem` – pointer to the KINSOL memory block.
- `constraints` – 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:

- `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.

int **KINSetSysFunc**(void *kin_mem, *KINSysFn* func)

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

Arguments:

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

Return value:

- `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.

int **KINSetReturnNewest**(void *kin_mem, *booleantype* ret_newest)

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` – pointer to the KINSOL memory block.
- `ret_newest` – `SUNTRUE` – return the newest iteration. `SUNFALSE` – return the iteration consistent with the last function evaluation.

Return value:

- 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.

int **KINSetDamping**(void *kin_mem, *realtype* beta)

The function *KINSetDamping()* specifies the value of the damping parameter in the fixed point or Picard iteration.

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- beta – the damping parameter value $0 < beta \leq 1.0$.

Return value:

- 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.

int **KINSetMAA**(void *kin_mem, long int maa)

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

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- maa – subspace size for various methods. A value of 0 means no acceleration, while a positive value means acceleration will be done.

Return value:

- 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.

int **KINSetDampingAA**(void *kin_mem, *realtype* beta)

The function *KINSetDampingAA()* specifies the value of the Anderson acceleration damping paramter.

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- beta – the damping parameter value $0 < beta \leq 1.0$.

Return value:

- 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.

int **KINSetDelayAA**(void *kin_mem, long int delay)

The function *KINSetDelayAA()* specifies the number of iterations to delay the start of Anderson acceleration.

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- delay – the number of iterations to delay Anderson acceleration.

Return value:

- 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.

int **KINSetOrthAA**(void *kin_mem, int orthaa)

The function *KINSetOrthAA()* specifies the orthogonalization routine to be used in the QR factorization portion of Anderson acceleration.

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- **orthaa – the orthogonalization routine parameter. Can be set to any of** the following
  - KIN_ORTH_MGS – Modified Gram Schmidt (default)
  - KIN_ORTH_ICWY – Inverse Compact WY Modified Gram Schmidt
  - KIN_ORTH_CGS2 – Classical Gram Schmidt with Reorthogonalization (CGS2)
  - KIN_ORTH_DCGS2 – Classical Gram Schmidt with Delayed Reorthogonlization

Return value:

- KIN_SUCCESS – The optional value has been successfully set.
- KIN_MEM_NULL – The kin_mem pointer is NULL.
- KIN_ILL_INPUT – The argument orthaa was not one of the predefined orthogonalization routines defined in KINSOL.

Note: This function *must* be called before calling *KINInit()*.

An example of how to use this function can be found in examples/kinsol/serial/kinAnalytic_fp.c

---

5.5.4.1 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 the *SUNMATRIX_DENSE* or *SUNMATRIX_BAND* modules for 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()*.

int **KINSetJacFn**(void *kin_mem, *KINLsJacFn* jac)
The function *KINSetJacFn()* specifies the Jacobian approximation function to be used for a matrix-based solver within the KINLS interface.

Arguments:

- `kin_mem` – pointer to the KINSOL solver object.
- `jac` – user-defined Jacobian approximation function. See *KINLsJacFn* for more details.

Return value:

- `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: This function must be called after the KINLS linear solver interface has been initialized through a call to *KINSetLinearSolver()*. By default, KINLS uses an internal difference quotient function for the *SUNMATRIX_DENSE* and *SUNMATRIX_BAND* modules. If `NULL` is passed to `jac`, this default function is used. An error will occur if no `jac` is supplied when using other matrix types.

Warning: 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 §5.6.5 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 `jtimes` each time it is called. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied functions without using global data in the program.

int **KINSetJacTimesVecFn**(void *kin_mem, *KINLsJacTimesVecFn* jtimes)
The function *KINSetJacTimesVecFn()* specifies the Jacobian-vector product function.

Arguments:

- `kin_mem` – pointer to the KINSOL memory block.
- `jtimes` – user-defined Jacobian-vector product function.

Return value:

- `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 §5.6.5. 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 [13] 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.

int **KINSetJacTimesVecSysFn**(void *kin_mem, *KINSysFn* jtimesSysFn)

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

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- jtimesSysFn – is the CC 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 §5.6.1.)

Return value:

- 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()*.

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 §5.6. 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.

int **KINSetPreconditioner**(void *kin_mem, *KINLsPrecSetupFn* psetup, *KINLsPrecSolveFn* psolve)

The function *KINSetPreconditioner()* specifies the preconditioner setup and solve functions.

Arguments:

- kin_mem – pointer to the KINSOL solver object.

- `psetup` – user-defined function to set up the preconditioner. See *KINLsPrecSetupFn* for more details. Pass `NULL` if no setup is necessary.
- `psolve` – user-defined preconditioner solve function. See *KINLsPrecSolveFn* for more details.

Return value:

- `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 `SUNLinearSolver` 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()*.

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

5.5.5 Optional output functions

KINSOL provides an extensive list of functions that can be used to obtain solver performance information. Table 5.2 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`).

Table 5.2: 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	*KINGetNumNonlinSolvIters()*
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()*

5.5.5.1 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.

int **KINGetWorkSpace**(void *kin_mem, long int lenrw, long int leniw)
The function *KINGetWorkSpace()* returns the KINSOL integer and real workspace sizes.

Arguments:

- `kin_mem` – pointer to the KINSOL memory block.
- `lenrw` – the number of `realtype` values in the KINSOL workspace.
- `leniw` – the number of integer values in the KINSOL workspace.

Return value:

- `KIN_SUCCESS` – The optional output values have been successfully set.
- `KIN_MEM_NULL` – The `kin_mem` pointer is `NULL`.

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

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

int **KINGetNumFuncEvals**(void *kin_mem, long int nfevals)
The function *KINGetNumFuncEvals()* returns the number of evaluations of the system function.

Arguments:

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

Return value:

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

int **KINGetNumNonlinSolvIters**(void *kin_mem, long int nniters)
The function *KINGetNumNonlinSolvIters()* returns the number of nonlinear iterations.

Arguments:

- `kin_mem` – pointer to the KINSOL memory block.
- `nniters` – number of nonlinear iterations.

Return value:

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

int **KINGetNumBetaCondFails**(void *kin_mem, long int nbcfails)
The function *KINGetNumBetaCondFails()* returns the number of β-condition failures.

Arguments:

- `kin_mem` – pointer to the KINSOL memory block.
- `nbcfails` – number of β -condition failures.

Return value:

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

int **KINGetNumBacktrackOps**(void *kin_mem, long int nbacktr)
The function *KINGetNumBacktrackOps()* returns the number of backtrack operations (step length adjustments) performed by the line search algorithm.

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- nbacktr – number of backtrack operations.

Return value:

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

int **KINGetFuncNorm**(void *kin_mem, *realtype* fnorm)
The function *KINGetFuncNorm()* returns the scaled Euclidean ℓ_2 norm of the nonlinear system function $F(u)$ evaluated at the current iterate.

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- fnorm – current scaled norm of $F(u)$.

Return value:

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

int **KINGetStepLength**(void *kin_mem, *realtype* steplength)
The function *KINGetStepLength()* returns the scaled Euclidean ℓ_2 norm of the step used during the previous iteration.

Arguments:

- kin_mem – pointer to the KINSOL memory block.
- steplength – scaled norm of the Newton step.

Return value:

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

5.5.5.2 KINLS linear solver interface optional output functions

The following optional outputs are available from the KINLS modules:

int **KINGetLinWorkSpace**(void *kin_mem, long int *lenrwLS, long int *leniwLS)
The function *KINGetLinWorkSpace()* returns the sizes of the real and integer workspaces used by the KINLS linear solver interface.

Arguments:

- kin_mem – pointer to the KINSOL solver object.
- lenrwLS – the number of real values in the KINLS workspace.

- leniwLS – the number of integer values in the KINLS workspace.

Return value:

- 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 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 SUNLinearSolver object attached to it. The template Jacobian matrix allocated by the user outside of KINLS is not included in this report.

Warning: The previous routines KINDlsGetWorkspace() and KINSpilsGetWorkspace() are now deprecated.

int **KINGetNumJacEvals**(void *kin_mem, long int *njevals)

The function *KINGetNumJacEvals()* returns the cumulative number of calls to the KINLS Jacobian approximation function.

Arguments:

- kin_mem – pointer to the KINSOL solver object.
- njevals – the cumulative number of calls to the Jacobian function total so far.

Return value:

- 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 has not been initialized.

Warning: The previous routine KINDlsGetNumJacEvals() is now deprecated,

int **KINGetNumLinFuncEvals**(void *kin_mem, long int *nrevalsLS)

The function KINGetNumLinResEvals() returns the cumulative number of calls to the user residual function due to the finite difference Jacobian approximation or finite difference Jacobian-vector product approximation.

Arguments:

- kin_mem – pointer to the KINSOL solver object.
- nrevalsLS – the cumulative number of calls to the user residual function.

Return value:

- 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 has not been initialized.

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

Warning: The previous routines KINDlsGetNumRhsEvals() and KINSpilsGetNumRhsEvals() are now deprecated.

int **KINGetNumLinIters**(void *kin_mem, long int *nliters)
The function *KINGetNumLinIters()* returns the cumulative number of linear iterations.

Arguments:

- kin_mem – pointer to the KINSOL solver object.
- nliters – the current number of linear iterations.

Return value:

- 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 has not been initialized.

Warning: The previous routine KINSpilsGetNumLinIters() is now deprecated.

int **KINGetNumLinConvFails**(void *kin_mem, long int *nlcfails)
The function *KINGetNumLinConvFails()* returns the cumulative number of linear convergence failures.

Arguments:

- kin_mem – pointer to the KINSOL solver object.
- nlcfails – the current number of linear convergence failures.

Return value:

- 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 has not been initialized.

Warning: The previous routine KINSpilsGetNumConvFails() is now deprecated.

int **KINGetNumPrecEvals**(void *kin_mem, long int *npevals)
The function *KINGetNumPrecEvals()* returns the cumulative number of preconditioner evaluations, i.e., the number of calls made to psetup.

Arguments:

- kin_mem – pointer to the KINSOL solver object.
- npevals – the cumulative number of calls to psetup.

Return value:

- 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 has not been initialized.

Warning: The previous routine KINSpilsGetNumPrecEvals() is now deprecated.

int **KINGetNumPrecSolves**(void *kin_mem, long int *npsolves)
The function *KINGetNumPrecSolves()* returns the cumulative number of calls made to the preconditioner solve function, `psolve`.

Arguments:

- `kin_mem` – pointer to the KINSOL solver object.
- `npsolves` – the cumulative number of calls to `psolve`.

Return value:

- `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 has not been initialized.

Warning: The previous routine `KINSpilsGetNumPrecSolves()` is now deprecated.

int **KINGetNumJtimesEvals**(void *kin_mem, long int *njvevals)
The function *KINGetNumJtimesEvals()* returns the cumulative number of calls made to the Jacobian-vector product function, `jtimes`.

Arguments:

- `kin_mem` – pointer to the KINSOL solver object.
- `njvevals` – the cumulative number of calls to `jtimes`.

Return value:

- `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 has not been initialized.

Warning: The previous routine `KINSpilsGetNumJtimesEvals()` is now deprecated.

int **KINGetLastLinFlag**(void *kin_mem, long int *lsflag)
The function *KINGetLastLinFlag()* returns the last return value from an KINLS routine.

Arguments:

- `kin_mem` – pointer to the KINSOL solver object.
- `lsflag` – the value of the last return flag from an KINLS function.

Return value:

- `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 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 SUNLinearSolver object, then lsflag will be SUNLS_PSET_FAIL_UNREC, SUNLS_ASET_FAIL_UNREC, or SUNLS_PACKAGE_FAIL_UNREC.

If the KINLS solve function failed (KINSolve() returned KIN_LSOLVE_FAIL), lsflag contains the error return flag from the SUNLinearSolver object, which will be one of: SUNLS_MEM_NULL, indicating that the SUNLinearSolver memory is NULL; SUNLS_ATIMES_FAIL_UNREC, indicating an unrecoverable failure in the $J * v$ function; SUNLS_PSOLVE_FAIL_UNREC, indicating that the preconditioner solve function psolve failed unrecoverably; 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.

Warning: The previous routines KINDlsGetLastFlag() and KINSpilsGetLastFlag() are now deprecated.

char ***KINGetLinReturnFlagName**(long int lsflag)

The function *KINGetLinReturnFlagName()* returns the name of the KINLS constant corresponding to lsflag.

Arguments:

- flag – the flag returned by a call to an KINSOL function

Return value:

- char* – the flag name string or if $1 \leq$ lsflag $\leq N$ (LU factorization failed), this function returns "NONE".

Warning: The previous routines KINDlsGetReturnFlagName() and KINSpilsGetReturnFlagName() are now deprecated.

5.6 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.

5.6.1 Problem defining function

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

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

This function computes the $F(u)$ (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 dependent variable vector, u
- fval – is the output vector $F(u)$
- user_data – is a pointer to user data, the same as the user_data pointer parameter passed to *KINSetUserData()*

Return value: An *KINSysFn* function type should return a value of 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if a nonrecoverable error occurred. In the last case, the integrator halts. If a recoverable error occurred, the integrator will attempt to correct and retry.

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

5.6.2 Error message handler function

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

typedef void (***KINErrHandlerFn**)(int error_code, const char *module, const char *function, char *msg, void *user_data)

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
- `eH_data` – is a pointer to user data, the same as the `eh_data` parameter passed to *KINSetErrHandlerFn()*

Return value: This 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.

5.6.3 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:

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

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
- `ih_data` – is a pointer to user data, the same as the `ih_data` parameter passed to *KINSetInfoHandlerFn()*

Return value: This function has no return value.

5.6.4 Jacobian construction (matrix-based linear solvers)

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

typedef int (***KINLsJacFn**)(*N_Vector* u, *N_Vector* fu, *SUNMatrix* J, void *user_data, *N_Vector* tmp1, *N_Vector* tmp2)

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 (of type SUNMatrix), $F'(u)$.
- 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 *KINLsJacFn* function as temporary storage or work space.

Return value: An *KINLsJacFn* should return 0 if successful, or a non-zero value otherwise.

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

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$= 0 \ldots$ N-1). This macro is meant for small problems for which efficiency of access is not a major concern. Thus, in terms of the indices m and n ranging from 1 to N, the Jacobian element $J_{m,n}$ can be set using the statement SM_ELEMENT_D(J, m-1, n-1) = $J_{m,n}$. Alternatively, SM_COLUMN_D(J, j) returns a pointer to the first element of the j-th column of J (with j$= 0 \ldots$ N-1), and the elements of the j-th column can then be accessed using ordinary array indexing. Consequently, $J_{m,n}$ can be loaded using the statements col_n = SM_COLUMN_D(J, n-1); col_n[m-1] = $J_{m,n}$. For large problems, it is more efficient to use SM_COLUMN_D than to use SM_ELEMENT_D. Note that both of these macros number rows and columns starting from 0. The SUNMATRIX_DENSE type and accessor macros are documented in §7.3.

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 -mupper $\leq$ m-n $\leq$ 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 $-$mupper to mlower. For large problems, it is more efficient to use SM_COLUMN_B and SM_COLUMN_ELEMENT_B than to use the SM_ELEMENT_B macro. As in the dense case, these macros all number rows and columns starting from 0. The SUNMATRIX_BAND type and accessor macros are documented in §7.6.

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 §7.8.

Warning: 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.

5.6.5 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 matrix-vector products Jv. If such a function is not supplied, the default is a difference quotient approximation to these products.

typedef int (***KINLsJacTimesVecFn**)(*N_Vector* v, *N_Vector* Jv, *N_Vector* u, *booleantype* *new_u, void *user_data)

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

Arguments:

- v – is the vector by which the Jacobian must be multplied to the right.
- Jv – is the computed output vector.
- u – is the current value of the dependent variable vector.
- 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`.

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

5.6.6 Preconditioner solve (iterative linear solvers)

If a user-supplied preconditioner is to be used with a `SUNLinearSolver` solver module, then the user must provide a function to solve the linear system $Pz = r$ where P is the preconditioner matrix which approximates (at least crudely) the Jacobian matrix $J = F'(u)$. This function must be of type *KINLsPrecSolveFn*, defined as follows:

typedef int (***KINLsPrecSolveFn**)(*N_Vector* u, *N_Vector* uscale, *N_Vector* fval, *N_Vector* fscale, *N_Vector* v, void *user_data)

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 `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 inpuut, `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 *KINSetUserData()*.

Return value: The value returned by the preconditioner solve function should be 0 if successful, positive for a recoverable error, or 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.

5.6.7 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:

typedef int (***KINLsPrecSetupFn**)(*N_Vector* u, *N_Vector* uscale, *N_Vector* fval, *N_Vector* fscale, void *user_data)

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 `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 *KINSetUserData()*.

Return value: The value returned by the preconditioner setup function should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

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

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

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.

5.7 A parallel band-block-diagonal preconditioner module

The efficiency of Krylov iterative methods for the solution of linear systems can be greatly enhanced through preconditioning. For problems in which the user cannot define a more effective, problem-specific preconditioner, KINSOL provides a band-block-diagonal preconditioner module KINBBDPRE, to be used with the parallel `N_Vector` module described in §6.5.

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

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

The KINBBDPRE module calls two user-provided functions to construct P: a required function `Gloc` (of type *KINBBDLocalFn*) which approximates the nonlinear system function $G(u) \approx F(u)$ 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.

typedef int (***KINBBDLocalFn**)(*sunindextype* Nlocal, *N_Vector* u, *N_Vector* gval, void *user_data)

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: An *KINBBDLocalFn* function type should return 0 to indicate success, or non-zero if an error occured.

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

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

typedef int (***KINBBDCommFn**)(*sunindextype* Nlocal, *N_Vector* u, void *user_data)

This `Gcomm` function performs all inter-processor communications necessary for the execution of the `Gloc` function above, using the input vectors `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: An *KINBBDLocalFn* function type should return 0 to indicate success, or non-zero if an error occured.

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 residual 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()*.

Besides the header files required for the integration of the DAE problem (see §5.3), to use the KINBBDPRE module, the main program must include the header file `kin_bbdpre.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 §5.4 are not bold.

1. Initialize parallel or multi-threaded environment (*if appropriate*)
2. Create the SUNDIALS context object
3. Set the problem dimensions etc.
4. Create the vector with the initial guess
5. Create matrix object (*if appropriate*)

6. **Create linear solver object** (*if appropriate*)

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

7. Create nonlinear solver object (*if appropriate*)
8. Create KINSOL object
9. Initialize KINSOL solver
10. Attach the linear solver (*if appropriate*)
11. **Set linear solver optional inputs** (*if appropriate*)

    Note that the user should not overwrite the preconditioner setup function or solve function through calls to *KINSetPreconditioner()* optional input function.

12. **Initialize the KINBBDPRE preconditioner module**

    Call *KINBBDPrecInit()* to allocate memory and initialize the internal preconditioner data. The last two arguments of *KINBBDPrecInit()* are the two user-supplied functions described above.

13. Set optional inputs
14. Solve problem
15. **Get optional outputs**

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

16. Deallocate memory
17. Finalize MPI, if used

The user-callable functions that initialize or re-initialize the KINBBDPRE preconditioner module are described next.

int **KINBBDPrecInit**(void *kin_mem, *sunindextype* Nlocal, *sunindextype* mudq, sunindexype mldq, *sunindextype* mukeep, *sunindextype* mlkeep, *realtype* dq_rel_u, *KINBBDLocalFn* Gloc, *KINBBDCommFn* Gcomm)

The function *KINBBDPrecInit()* initializes and allocates memory for the KINBBDPRE preconditioner.

Arguments:

- `kin_mem` – pointer to the KINSOL memory block.
- `Nlocal` – local vector length.
- `mudq` – upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
- `mldq` – lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
- `mukeep` – upper half-bandwidth of the retained banded approximate Jacobian block.
- `mlkeep` – lower half-bandwidth of the retained banded approximate Jacobian block.
- `dq_rel_u` – the relative increment in components of `u` used in the difference quotient approximations. The default is `dq_rel_u` $= \sqrt{\text{unit roundoff}}$, which can be specified by passing `dq_rel_u= 0.0`.
- `Gloc` – the CC function which computes the approximation $G(u) \approx F(u)$.
- `Gcomm` – the optional CC function which performs all interprocess communication required for the computation of $G(u)$.

Return value:

- `KINLS_SUCCESS` – The call to *KINBBDPrecInit()* was successful.

- KINLS_MEM_NULL – The kin_mem pointer was NULL.
- KINLS_MEM_FAIL – A memory allocation request has failed.
- KINLS_LMEM_NULL – The KINLS linear solver interface has not been initialized.
- KINLS_ILL_INPUT – The supplied vector implementation was not compatible with the block band preconditioner.

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

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

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

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

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

int **KINBBDPrecGetWorkSpace**(void *kin_mem, long int *lenrwBBDP, long int *leniwBBDP)

The function *KINBBDPrecGetWorkSpace()* returns the local sizes of the KINBBDPRE real and integer workspaces.

Arguments:

- kin_mem – pointer to the KINSOL solver object.
- lenrwBBDP – local number of real values in the KINBBDPRE workspace.
- leniwBBDP – local number of integer values in the KINBBDPRE workspace.

Return value:

- 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 SUNLinearSolver 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.

int **KINBBDPrecGetNumGfnEvals**(void *kin_mem, long int *ngevalsBBDP)

The function *KINBBDPrecGetNumGfnEvals()* returns the cumulative number of calls to the user Gres function due to the finite difference approximation of the Jacobian blocks used within KINBBDPRE's preconditioner setup function.

Arguments:

- kin_mem – pointer to the KINSOL solver object.
- ngevalsBBDP – the cumulative number of calls to the user Gres function.

Return value:

- 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` evaluations of `Gres`, the costs associated with KINBBDPRE also includes `nlinsetups` LU factorizations, `nlinsetups` calls to `Gcomm`, `npsolves` banded backsolve calls, and `nrevalsLS` residual function evaluations, where `nlinsetups` is an optional KINSOL output (see §5.5.5.1), and `npsolves` and `nrevalsLS` are linear solver optional outputs (see §5.5.5.2).

5.8 Alternative to KINSOL for difficult systems

A nonlinear system $F(u) = 0$ may be difficult to solve with KINSOL (or any other nonlinear system solver) for a variety of reasons. The possible reasons include high nonlinearity, small region of convergence, and lack of a good initial guess. For systems with such difficulties, there is an alternative approach that may be more successful. This is an old idea, but deserves some new attention.

If the nonlinear system is $F(u) = 0$, consider instead the ODE system $du/dt = -M^{-1}F(u)$, where M is a nonsingular matrix that is an approximation (even a crude approximation) to the system Jacobian $F_u = dF/du$. Whatever M is, if this ODE is solved until it reaches a steady state u^*, then u^* is a zero of the right-hand side of the ODE, and hence a solution to $F(u) = 0$. There is no issue of having a close enough initial guess.

A further basis for this choice of ODE is the following: If M approximates F_u, then the Jacobian of the ODE system, $-M^{-1}F_u$, is approximately equal to $-I$ where I is the identity matrix. This means that (in a local approximation sense) the solution modes of the ODE behave like $\exp(-t)$, and that asymptotically the approach to the steady state goes as $\exp(-t)$. Of course, the closer M is to F_u, the better this basis applies.

Using (say) CVODE to solve the above ODE system requires, in addition to the objective function $F(u)$, the calculation of a suitable matrix M and its inverse, or at least a routine that solves linear systems $Mx = b$. This is similar to the KINSOL requirement of supplying the system Jacobian J (or solutions to $Jx = b$), but differs in that M may be simpler than J and hence easier to deal with. Depending on the nature of M, this may be handled best with a direct solver, or with a preconditioned Krylov solver. The latter calls for the use of a preconditioner P that may be a crude approximation to M, hence even easier to solve. Note if using ARKODE, the ODE system may be posed in the linearly implicit from $Mdu/dt = -F(u)$ where M is the "mass matrix" for the system. This use case requires supplying ARKODE with a function to evaluate M or to compute its action on a vector ($Mv = w$) and attaching a linear solver (direct or iterative) to solve the linear systems $Mx = b$.

The solution of the ODE may be made easier by solving instead the equivalent DAE, $Mdu/dt + F(u) = 0$. Applying IDA to this system requires solutions to linear systems whose matrix is the DAE system Jacobian, $J = F_u + \alpha M$, where α is the scalar coefficient c_j supplied to the user's Jacobian or preconditioner routine. Selecting a preconditioned Krylov method requires an approximation to this Jacobian as preconditioner P. Given that M approximates F_u (possibly crudely), the appropriate approximation to J is $P = M + \alpha M = (1 + \alpha)M$. Again the user must supply a routine that solves linear systems $Px = b$, or $Mx = b/(1 + \alpha)$. If M is too difficult to solve, than an approximation M' that is easier can be substituted, as long as it achieves convergence. As always, there is a trade-off between the expense of solving M and the difficulty of achieving convergence in the linear solver.

For the solution of either the ODE or DAE system above, the chances for convergence can be improved with a piecewise constant choice for M. Specifically, starting from an initial guess u_0, an initial choice for M might be $M_0 = F_u(u_0)$, or some approximation to this Jacobian. Then one could integrate $M_0 du/dt + F(u) = 0$ from $t = 0$ to $t = T$ for some sizable value T, evaluate $F_u(u(T))$, and take M_1 to be an approximation to that Jacobian. Then integrate using M_1 from $t = T$ to $t = 2T$, and repeat the process until it converges to a steady state.

Chapter 6

Vector Data Structures

The SUNDIALS library comes packaged with a variety of NVECTOR implementations, designed for simulations in serial, shared-memory parallel, and distributed-memory parallel environments, as well as interfaces to vector data structures used within external linear solver libraries. All native implementations assume that the process-local data is stored contiguously, and they in turn provide a variety of standard vector algebra operations that may be performed on the data.

In addition, SUNDIALS provides a simple interface for generic vectors (akin to a C++ *abstract base class*). All of the major SUNDIALS solvers (CVODE(s), IDA(s), KINSOL, ARKODE) in turn are constructed to only depend on these generic vector operations, making them immediately extensible to new user-defined vector objects. The only exceptions to this rule relate to the dense, banded and sparse-direct linear system solvers, since they rely on particular data storage and access patterns in the NVECTORS used.

6.1 Description of the NVECTOR Modules

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, and specific to, the particular NVECTOR implementation. Users can provide a custom implementation of the NVECTOR module or use one provided within SUNDIALS. The generic operations are described below. In the sections following, the implementations provided with SUNDIALS are described.

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**

and the generic structure is defined as

```
struct _generic_N_Vector {
   void *content;
   struct _generic_N_Vector_Ops *ops;
};
```

Here, the `_generic_N_Vector_Op` structure is essentially a list of function 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);
```

(continues on next page)

(continued from previous page)

```
  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     (*nvl1normlocal)(N_Vector);
  booleantype  (*nvinvtestlocal)(N_Vector, N_Vector);
  booleantype  (*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          (*nvdotprodmultilocal)(int, N_Vector, N_Vector *, realtype *);
  int          (*nvdotprodmultiallreduce)(int, N_Vector, realtype *);
```

(continues on next page)

(continued from previous page)

```
   int          (*nvbufsize)(N_Vector, sunindextype *);
   int          (*nvbufpack)(N_Vector, void*);
   int          (*nvbufunpack)(N_Vector, void*);
};
```

The generic NVECTOR module defines and implements the vector operations acting on a `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 operation $z \leftarrow cx$ for vectors x and z and a scalar c:

```
void N_VScale(realtype c, N_Vector x, N_Vector z) {
   z->ops->nvscale(c, x, z);
}
```

§6.2 contains a complete list of all standard vector operations defined by the generic NVECTOR module. §6.2.2, §6.2.3, §6.2.4, §6.2.5, and §6.2.6 list *optional* fused, vector array, local reduction, single buffer reduction, and exchange operations, respectively.

Fused and vector array operations (see §6.2.2 and §6.2.3) 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.

Local reduction operations (see §6.2.4) are similarly intended to reduce parallel communication on distributed memory systems, particularly when NVECTOR objects are combined together within an NVECTOR_MANYVECTOR object (see §6.16). If a particular NVECTOR implementation defines a local reduction operation as `NULL`, the NVECTOR_MANYVECTOR 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 implementations.

The single buffer reduction operations (§6.2.5) are used in low-synchronization methods to combine separate reductions into one `MPI_Allreduce` call.

The exchange operations (see §6.2.6) are intended only for use with the XBraid library for parallel-in-time integration (accessible from ARKODE) and are otherwise unused by SUNDIALS packages.

6.1.1 NVECTOR Utility Functions

The generic NVECTOR module also defines several utility functions to aid in creation and management of arrays of `N_Vector` objects – these functions are particularly useful for Fortran users to utilize the NVECTOR_MANYVECTOR or SUNDIALS' sensitivity-enabled packages CVODES and IDAS.

The functions *N_VCloneVectorArray()* and *N_VCloneVectorArrayEmpty()* create (by cloning) an array of *count* variables of type *N_Vector*, each of the same type as an existing `N_Vector` input:

N_Vector ***N_VCloneVectorArray**(int count, *N_Vector* w)
Clones an array of `count` `N_Vector` objects, allocating their data arrays (similar to *N_VClone()*).

Arguments:

- `count` – number of `N_Vector` objects to create.
- `w` – template *N_Vector* to clone.

Return value:

- pointer to a new `N_Vector` array on success.
- `NULL` pointer on failure.

N_Vector ***N_VCloneVectorArrayEmpty**(int count, *N_Vector* w)
Clones an array of `count` `N_Vector` objects, leaving their data arrays unallocated (similar to *N_VCloneEmpty()*).

Arguments:

- `count` – number of `N_Vector` objects to create.
- `w` – template *N_Vector* to clone.

Return value:

- pointer to a new `N_Vector` array on success.
- `NULL` pointer on failure.

An array of variables of type *N_Vector* can be destroyed by calling *N_VDestroyVectorArray()*:

void **N_VDestroyVectorArray**(*N_Vector* *vs, int count)
Destroys an array of `count` `N_Vector` objects.

Arguments:

- `vs` – `N_Vector` array to destroy.
- `count` – number of `N_Vector` objects in `vs` array.

Notes: This routine will internally call the `N_Vector` implementation-specific *N_VDestroy()* operation.

If `vs` was allocated using *N_VCloneVectorArray()* then the data arrays for each `N_Vector` object will be freed; if `vs` was allocated using *N_VCloneVectorArrayEmpty()* then it is the user's responsibility to free the data for each `N_Vector` object.

Finally, we note that users of the Fortran 2003 interface may be interested in the additional utility functions *N_VNewVectorArray()*, *N_VGetVecAtIndexVectorArray()*, and *N_VSetVecAtIndexVectorArray()*, that are wrapped as `FN_NewVectorArray`, `FN_VGetVecAtIndexVectorArray`, and `FN_VSetVecAtIndexVectorArray`, respectively. These functions allow a Fortran 2003 user to create an empty vector array, access a vector from this array, and set a vector within this array:

N_Vector ***N_VNewVectorArray**(int count)
Creates an array of `count` `N_Vector` objects, the pointers to each are initialized as `NULL`.

Arguments:

- `count` – length of desired `N_Vector` array.

Return value:

- pointer to a new `N_Vector` array on success.
- `NULL` pointer on failure.

N_Vector ***N_VGetVecAtIndexVectorArray**(*N_Vector* *vs, int index)
Accesses the `N_Vector` at the location `index` within the `N_Vector` array `vs`.

Arguments:

- `vs` – `N_Vector` array.
- `index` – desired `N_Vector` to access from within `vs`.

Return value:

- pointer to the indexed `N_Vector` on success.
- `NULL` pointer on failure (`index < 0` or `vs == NULL`).

Notes: This routine does not verify that `index` is within the extent of `vs`, since `vs` is a simple `N_Vector` array that does not internally store its allocated length.

void **N_VSetVecAtIndexVectorArray**(*N_Vector* *vs, int index, *N_Vector* w)
Sets a pointer to `w` at the location `index` within the vector array `vs`.

Arguments:

- `vs` – `N_Vector` array.
- `index` – desired location to place the pointer to `w` within `vs`.
- `w` – `N_Vector` to set within `vs`.

Notes: This routine does not verify that `index` is within the extent of `vs`, since `vs` is a simple `N_Vector` array that does not internally store its allocated length.

6.1.2 Implementing a custom NVECTOR

A particular implementation of the NVECTOR module must:

- Specify the *content* field of the `N_Vector` structure.
- 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`.

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 that only required operations need to be set, and that all operations are copied when cloning a vector.

N_Vector **N_VNewEmpty**()
This allocates a new generic `N_Vector` object and initializes its content pointer and the function pointers in the operations structure to `NULL`.

Return value: If successful, this function returns an `N_Vector` object. If an error occurs when allocating the object, then this routine will return `NULL`.

void **N_VFreeEmpty**(*N_Vector* v)
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* – an N_Vector object

int **N_VCopyOps**(*N_Vector* w, *N_Vector* v)
This function copies the function pointers in the `ops` structure of `w` into the `ops` structure of `v`.

Arguments:

- *w* – the vector to copy operations from
- *v* – the vector to copy operations to

Return value: If successful, this function returns `0`. If either of the inputs are `NULL` or the `ops` structure of either input is `NULL`, then is function returns a non-zero value.

Each NVECTOR implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 6.1. It is recommended that a user supplied NVECTOR implementation use the `SUNDIALS_NVEC_CUSTOM` identifier.

Table 6.1: Vector Identifications associated with vector kernels supplied with SUNDIALS

Vector ID	Vector type	ID Value
SUNDIALS_NVEC_SERIAL	Serial	0
SUNDIALS_NVEC_PARALLEL	Distributed memory parallel (MPI)	1
SUNDIALS_NVEC_OPENMP	OpenMP shared memory parallel	2
SUNDIALS_NVEC_PTHREADS	PThreads shared memory parallel	3
SUNDIALS_NVEC_PARHYP	*hypre* ParHyp parallel vector	4
SUNDIALS_NVEC_PETSC	PETSc parallel vector	5
SUNDIALS_NVEC_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

6.1.3 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 the subsection §6.2 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., $m = \min_{0 \le i < n} \text{real}(x_i)$.
- `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., $z_i = c + 0j$ for $0 \le i < n$.
- `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 each package's linear solver interface (e.g., ARKLS or CVLS) may be used on complex-valued problems, none of the built-in SUNMatrix or SUNLinearSolver modules will work (all of the direct linear solvers must store complex-valued data, and all of the iterative linear solvers require *N_VDotProd()*). Hence a complex-valued user must provide custom linear solver modules for their problem. At a minimum this will consist of a custom SUNLinearSolver implementation (see §8.1.8), and optionally a custom SUNMatrix as well. The user should then attach these modules 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`.

6.2 Description of the NVECTOR operations

6.2.1 Standard vector operations

The standard vector operations defined by the generic `N_Vector` module are defined as follows. For each of these operations, we give the name, usage of the function, and a description of its mathematical operations below.

N_Vector_ID **N_VGetVectorID**(*N_Vector* w)
Returns the vector type identifier for the vector `w`. It is used to determine the vector implementation type (e.g. serial, parallel, ...) from the abstract `N_Vector` interface. Returned values are given in Table 6.1.

Usage:

```
id = N_VGetVectorID(w);
```

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

Usage:

```
v = N_VClone(w);
```

N_Vector **N_VCloneEmpty**(*N_Vector* w)
Creates a new `N_Vector` of the same type as an existing vector *w* and sets the *ops* field. It does not allocate storage for the new vector's data.

Usage:

```
v = N VCloneEmpty(w);
```

void **N_VDestroy**(*N_Vector* v)
Destroys the `N_Vector` *v* and frees memory allocated for its internal data.

Usage:

```
N_VDestroy(v);
```

void **N_VSpace**(*N_Vector* v, *sunindextype* *lrw, *sunindextype* *liw)
Returns storage requirements for the `N_Vector` *v*:

- *lrw* contains the number of `realtype` words
- *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.

Usage:

```
N_VSpace(nvSpec, &lrw, &liw);
```

realtype ***N_VGetArrayPointer**(*N_Vector* v)
Returns a pointer to a `realtype` array from the `N_Vector` *v*. Note that this assumes that the internal data in the `N_Vector` is a contiguous array of `realtype` and is accesible from the CPU.

This routine is only used in the solver-specific interfaces to the dense and banded (serial) linear solvers, and in the interfaces to the banded (serial) and band-block-diagonal (parallel) preconditioner modules provided with SUNDIALS.

Usage:

```
vdata = N_VGetArrayPointer(v);
```

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

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

Usage:

```
vdata = N_VGetArrayPointer(v);
```

void **N_VSetArrayPointer**(*realtype* *vdata, *N_Vector* v)
Replaces the data array pointer in an `N_Vector` with a given array of `realtype`. Note that this assumes that the internal data in the `N_Vector` is a contiguous array of `realtype`. This routine is only used in the interfaces to the dense (serial) linear solver, hence need not exist in a user-supplied NVECTOR module.

Usage:

```
N_VSetArrayPointer(vdata,v);
```

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

Usage:

```
commptr = N_VGetCommunicator(v);
```

sunindextype **N_VGetLength**(*N_Vector* v)
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.

Usage:

```
global_length = N_VGetLength(v);
```

void **N_VLinearSum**(*realtype* a, *N_Vector* x, *realtype* b, *N_Vector* y, *N_Vector* z)
Performs the operation $z = ax + by$, where a and b are `realtype` scalars and x and y are of type `N_Vector`:

$$z_i = ax_i + by_i, \quad i = 0, \ldots, n-1.$$

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

Usage:

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

void **N_VConst**(*realtype* c, *N_Vector* z)
Sets all components of the `N_Vector` z to `realtype` c:

$$z_i = c, \quad i = 0, \ldots, n-1.$$

Usage:

```
N_VConst(c, z);
```

void **N_VProd**(*N_Vector* x, *N_Vector* y, *N_Vector* z)
Sets the `N_Vector` z to be the component-wise product of the `N_Vector` inputs x and y:

$$z_i = x_i y_i, \quad i = 0, \ldots, n-1.$$

Usage:

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

void **N_VDiv**(*N_Vector* x, *N_Vector* y, *N_Vector* z)
Sets the `N_Vector` z to be the component-wise ratio of the `N_Vector` inputs x and y:

$$z_i = \frac{x_i}{y_i}, \quad i = 0, \ldots, n-1.$$

The y_i may not be tested for 0 values. It should only be called with a y that is guaranteed to have all nonzero components.

Usage:

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

void **N_VScale**(*realtype* c, *N_Vector* x, *N_Vector* z)
Scales the `N_Vector` x by the `realtype` scalar c and returns the result in z:

$$z_i = cx_i, \quad i = 0, \ldots, n-1.$$

Usage:

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

void **N_VAbs**(*N_Vector* x, *N_Vector* z)
Sets the components of the `N_Vector` z to be the absolute values of the components of the `N_Vector` x:

$$z_i = |x_i|, \quad i = 0, \ldots, n-1.$$

Usage:

```
N_VAbs(x, z);
```

void **N_VInv**(*N_Vector* x, *N_Vector* z)

Sets the components of the `N_Vector` z to be the inverses of the components of the `N_Vector` x:

$$z_i = \frac{1}{x_i}, \quad i = 0, \ldots, n-1.$$

This routine may not check for division by 0. It should be called only with an x which is guaranteed to have all nonzero components.

Usage:

```
N_VInv(x, z);
```

void **N_VAddConst**(*N_Vector* x, *realtype* b, *N_Vector* z)

Adds the `realtype` scalar b to all components of x and returns the result in the `N_Vector` z:

$$z_i = x_i + b, \quad i = 0, \ldots, n-1.$$

Usage:

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

realtype **N_VDotProd**(*N_Vector* x, *N_Vector* z)

Returns the value of the dot-product of the `N_Vectors` x and y:

$$d = \sum_{i=0}^{n-1} x_i y_i.$$

Usage:

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

realtype **N_VMaxNorm**(*N_Vector* x)

Returns the value of the l_∞ norm of the `N_Vector` x:

$$m = \max_{0 \le i < n} |x_i|.$$

Usage:

```
m = N_VMaxNorm(x);
```

realtype **N_VWrmsNorm**(*N_Vector* x, *N_Vector* w)

Returns the weighted root-mean-square norm of the `N_Vector` x with (positive) `realtype` weight vector w:

$$m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i)^2\right) / n}$$

Usage:

```
m = N_VWrmsNorm(x, w);
```

realtype **N_VWrmsNormMask**(*N_Vector* x, *N_Vector* w, *N_Vector* id)

Returns the weighted root mean square norm of the `N_Vector` x with `realtype` weight vector w built using only the elements of x corresponding to 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 $H(\alpha) = \begin{cases} 1 & \alpha > 0 \\ 0 & \alpha \leq 0 \end{cases}$.

Usage:

```
m = N_VWrmsNormMask(x, w, id);
```

realtype **N_VMin**(*N_Vector* x)

Returns the smallest element of the `N_Vector` x:

$$m = \min_{0 \leq i < n} x_i.$$

Usage:

```
m = N_VMin(x);
```

realtype **N_VWl2Norm**(*N_Vector* x, *N_Vector* w)

Returns the weighted Euclidean l_2 norm of the `N_Vector` x with `realtype` weight vector w:

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

Usage:

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

realtype **N_VL1Norm**(*N_Vector* x)

Returns the l_1 norm of the `N_Vector` x:

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

Usage:

```
m = N_VL1Norm(x);
```

void **N_VCompare**(*realtype* c, *N_Vector* x, *N_Vector* z)

Compares the components of the `N_Vector` x to the `realtype` scalar c and returns an `N_Vector` z such that for all $0 \leq i < n$,

$$z_i = \begin{cases} 1.0 & \text{if } |x_i| \geq c, \\ 0.0 & \text{otherwise} \end{cases}.$$

Usage:

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

booleantype **N_VInvTest**(*N_Vector* x, *N_Vector* z)

Sets the components of the `N_Vector` z to be the inverses of the components of the `N_Vector` x, with prior testing for zero values:

$$z_i = \frac{1}{x_i}, \quad i = 0, \ldots, n-1.$$

This routine returns a boolean assigned to `SUNTRUE` if all components of x are nonzero (successful inversion) and returns `SUNFALSE` otherwise.

Usage:

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

booleantype **N_VConstrMask**(*N_Vector* c, *N_Vector* x, *N_Vector* m)
Performs the following constraint tests based on the values in c_i:

$$\begin{aligned} x_i &> 0 \quad \text{if} \quad c_i = 2, \\ x_i &\geq 0 \quad \text{if} \quad c_i = 1, \\ x_i &< 0 \quad \text{if} \quad c_i = -2, \\ x_i &\leq 0 \quad \text{if} \quad c_i = -1. \end{aligned}$$

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.

Usage:

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

realtype **N_VMinQuotient**(*N_Vector* num, *N_Vector* denom)
This routine returns the minimum of the quotients obtained by termwise dividing the elements of *n* by the elements in *d*:

$$\min_{0 \leq i < n} \frac{\text{num}_i}{\text{denom}_i}.$$

A zero element in *denom* will be skipped. If no such quotients are found, then the large value BIG_REAL (defined in the header file sundials_types.h) is returned.

Usage:

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

6.2.2 Fused operations

The following fused vector operations are *optional*. These operations are intended to increase data reuse, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. If a particular NVECTOR implementation defines one of the fused vector operations as NULL, the NVECTOR interface will call one of the above standard vector operations as necessary. As above, for each operation, we give the name, usage of the function, and a description of its mathematical operations below.

int **N_VLinearCombination**(int nv, *realtype* *c, *N_Vector* *X, *N_Vector* z)
This routine computes the linear combination of *nv* vectors with n elements:

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

where c is an array of nv scalars, x_j is a vector in the vector array *X*, and z is the output vector. If the output vector z is one of the vectors in *X*, then it *must* be the first vector in the vector array. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
retval = N_VLinearCombination(nv, c, X, z);
```

int **N_VScaleAddMulti**(int nv, *realtype* *c, *N_Vector* x, *N_Vector* *Y, *N_Vector* *Z)
This routine scales and adds one vector to *nv* vectors with n elements:

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

where c is an array of scalars, x is a vector, y_j is a vector in the vector array *Y*, and z_j is an output vector in the vector array *Z*. The operation returns 0 for success and a non-zero value otherwise.

Usage:

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

int **N_VDotProdMulti**(int nv, *N_Vector* x, *N_Vector* *Y, *realtype* *d)
This routine computes the dot product of a vector with *nv* vectors having n elements:

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

where *d* is an array of scalars containing the computed dot products, x is a vector, and y_j is a vector the vector array *Y*. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
retval = N_VDotProdMulti(nv, x, Y, d);
```

6.2.3 Vector array operations

The following vector array operations are also *optional*. As with the fused vector operations, these are intended to increase data reuse, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. If a particular NVECTOR implementation defines one of the fused or vector array operations as `NULL`, the NVECTOR interface will call one of the above standard vector operations as necessary. As above, for each operation, we give the name, usage of the function, and a description of its mathematical operations below.

int **N_VLinearSumVectorArray**(int nv, *realtype* a, *N_Vector* X, *realtype* b, *N_Vector* *Y, *N_Vector* *Z)
This routine computes the linear sum of two vector arrays of *nv* vectors with n elements:

$$z_{j,i} = a x_{j,i} + b y_{j,i}, \quad i = 0, \ldots, n-1 \quad j = 0, \ldots, nv-1,$$

where *a* and *b* are scalars, x_j and y_j are vectors in the vector arrays *X* and *Y* respectively, and z_j is a vector in the output vector array *Z*. The operation returns 0 for success and a non-zero value otherwise.

Usage:

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

int **N_VScaleVectorArray**(int nv, *realtype* *c, *N_Vector* *X, *N_Vector* *Z)
This routine scales each element in a vector of n elements in a vector array of *nv* vectors by a potentially different constant:

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

where *c* is an array of scalars, x_j is a vector in the vector array *X*, and z_j is a vector in the output vector array *Z*. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
retval = N_VScaleVectorArray(nv, c, X, Z);
```

int **N_VConstVectorArray**(int nv, *realtype* c, *N_Vector* *Z)
This routine sets each element in a vector of n elements in a vector array of *nv* vectors to the same value:

$$z_{j,i} = c, \quad i = 0, \ldots, n-1 \quad j = 0, \ldots, nv-1,$$

where c is a scalar and z_j is a vector in the vector array *Z*. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
retval = N_VConstVectorArray(nv, c, Z);
```

int **N_VWrmsNormVectorArray**(int nv, *N_Vector* *X, *N_Vector* *W, *realtype* *m)
This routine computes the weighted root mean square norm of each vector in a vector array:

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

where x_j is a vector in the vector array *X*, w_j is a weight vector in the vector array *W*, and *m* is the output array of scalars containing the computed norms. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
retval = N_VWrmsNormVectorArray(nv, X, W, m);
```

int **N_VWrmsNormMaskVectorArray**(int nv, *N_Vector* *X, *N_Vector* *W, *N_Vector* id, *realtype* *m)
This routine computes the masked weighted root mean square norm of each vector in a vector array:

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

where $H(id_i) = 1$ if $id_i > 0$ and is zero otherwise, x_j is a vector in the vector array *X*, w_j is a weight vector in the vector array *W*, *id* is the mask vector, and *m* is the output array of scalars containing the computed norms. The operation returns 0 for success and a non-zero value otherwise.

Usage:

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

int **N_VScaleAddMultiVectorArray**(int nv, int nsum, *realtype* *c, *N_Vector* *X, *N_Vector* **YY, *N_Vector* **ZZ)
This routine scales and adds a vector array of *nv* vectors to *nsum* other vector arrays:

$$z_{k,j,i} = c_k x_{j,i} + y_{k,j,i}, \quad i = 0, \ldots, n-1 \quad j = 0, \ldots, nv-1, \quad k = 0, \ldots, nsum-1$$

where *c* is an array of scalars, x_j is a vector in the vector array *X*, $y_{k,j}$ is a vector in the array of vector arrays *YY*, and $z_{k,j}$ is an output vector in the array of vector arrays *ZZ*. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
retval = N_VScaleAddMultiVectorArray(nv, nsum, c, x, YY, ZZ);
```

int **N_VLinearCombinationVectorArray**(int nv, int nsum, *realtype* *c, *N_Vector* **XX, *N_Vector* *Z)
This routine computes the linear combination of *nsum* vector arrays containing *nv* vectors:

$$z_{j,i} = \sum_{k=0}^{nsum-1} c_k x_{k,j,i}, \quad i = 0, \ldots, n-1 \quad j = 0, \ldots, nv-1,$$

where *c* is an array of scalars, $x_{k,j}$ is a vector in array of vector arrays *XX*, and $z_{j,i}$ is an output vector in the vector array *Z*. If the output vector array is one of the vector arrays in *XX*, it *must* be the first vector array in *XX*. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
retval = N_VLinearCombinationVectorArray(nv, nsum, c, XX, Z);
```

6.2.4 Local reduction operations

The following local reduction operations are also *optional*. As with the fused and vector array operations, these are intended to reduce parallel communication on distributed memory systems. If a particular NVECTOR implementation defines one of the local reduction operations as `NULL`, the NVECTOR interface will call one of the above standard vector operations as necessary. As above, for each operation, we give the name, usage of the function, and a description of its mathematical operations below.

realtype **N_VDotProdLocal**(*N_Vector* x, *N_Vector* y)

This routine computes the MPI task-local portion of the ordinary dot product of x and 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).

Usage:

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

realtype **N_VMaxNormLocal**(*N_Vector* x)

This routine computes the MPI task-local portion of the maximum norm of the NVECTOR x:

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

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

Usage:

```
m = N_VMaxNormLocal(x);
```

realtype **N_VMinLocal**(*N_Vector* x)

This routine computes the smallest element of the MPI task-local portion of the NVECTOR x:

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

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

Usage:

```
m = N_VMinLocal(x);
```

realtype **N_VL1NormLocal**(*N_Vector* x)

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).

Usage:

```
n = N_VL1NormLocal(x);
```

realtype **N_VWSqrSumLocal**(*N_Vector* x, *N_Vector* w)

This routine computes the MPI task-local portion of the weighted squared sum of the NVECTOR *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).

Usage:

```
s = N_VWSqrSumLocal(x, w);
```

realtype **N_VWSqrSumMaskLocal**(*N_Vector* x, *N_Vector* w, *N_Vector* id)

This routine computes the MPI task-local portion of the weighted squared sum of the NVECTOR *x* with weight vector *w* built using only the elements of *x* corresponding to positive elements of the NVECTOR *id*:

$$m = \sum_{i=0}^{n_{local}-1} (x_i w_i H(id_i))^2,$$

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).

Usage:

```
s = N_VWSqrSumMaskLocal(x, w, id);
```

booleantype **N_VInvTestLocal**(*N_Vector* x)

This routine sets the MPI task-local components of the NVECTOR *z* to be the inverses of the components of the NVECTOR *x*, with prior testing for zero values:

$$z_i = \frac{1}{x_i},\ i = 0, \ldots, n_{local} - 1$$

where n_{local} corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications). This routine returns a boolean assigned to `SUNTRUE` if all task-local components of *x* are nonzero (successful inversion) and returns `SUNFALSE` otherwise.

Usage:

```
t = N_VInvTestLocal(x);
```

booleantype **N_VConstrMaskLocal**(*N_Vector* c, *N_Vector* x, *N_Vector* m)

Performs the following constraint tests based on the values in c_i:

$$\begin{aligned} x_i &> 0 \quad \text{if} \quad c_i = 2, \\ x_i &\geq 0 \quad \text{if} \quad c_i = 1, \\ x_i &< 0 \quad \text{if} \quad c_i = -2, \\ x_i &\leq 0 \quad \text{if} \quad c_i = -1. \end{aligned}$$

for all MPI task-local components of the vectors. This routine returns a boolean assigned to SUNFALSE if any task-local 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.

Usage:

```
t = N_VConstrMaskLocal(c, x, m);
```

realtype **N_VMinQuotientLocal**(*N_Vector* num, *N_Vector* denom)
This routine returns the minimum of the quotients obtained by term-wise dividing num_i by $denom_i$, 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.

Usage:

```
minq = N_VMinQuotientLocal(num, denom);
```

6.2.5 Single Buffer Reduction Operations

The following *optional* operations are used to combine separate reductions into a single MPI call by splitting the local computation and communication into separate functions. These operations are used in low-synchronization orthogonalization methods to reduce the number of MPI Allreduce calls. If a particular NVECTOR implementation does not define these operations additional communication will be required.

int **N_VDotProdMultiLocal**(int nv, *N_Vector* x, *N_Vector* *Y, *realtype* *d)
This routine computes the MPI task-local portion of the dot product of a vector x with *nv* vectors y_j:

$$d_j = \sum_{i=0}^{n_{local}-1} x_i y_{j,i}, \quad j = 0, \ldots, nv-1,$$

where d is an array of scalars containing the computed dot products, x is a vector, y_j is a vector in the vector array *Y*, and n_{local} corresponds to the number of components in the vector on this MPI task. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
retval = N_VDotProdMultiLocal(nv, x, Y, d);
```

int **N_VDotProdMultiAllReduce**(int nv, *N_Vector* x, *realtype* *d)
This routine combines the MPI task-local portions of the dot product of a vector x with *nv* vectors:

```
retval = MPI_Allreduce(MPI_IN_PLACE, d, nv, MPI_SUNREALTYPE, MPI_SUM, comm)
```

where *d* is an array of *nv* scalars containing the local contributions to the dot product and *comm* is the MPI communicator associated with the vector *x*. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
retval = N_VDotProdMultiAllReduce(nv, x, d);
```

6.2.6 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.

int **N_VBufSize**(*N_Vector* x, *sunindextype* *size)
This routine returns the buffer size need to exchange in the data in the vector *x* between computational nodes.

Usage:

```
flag = N_VBufSize(x, &buf_size)
```

int **N_VBufPack**(*N_Vector* x, void *buf)
This routine fills the exchange buffer *buf* with the vector data in *x*.

Usage:

```
flag = N_VBufPack(x, &buf)
```

int **N_VBufUnpack**(*N_Vector* x, void *buf)
This routine unpacks the data in the exchange buffer *buf* into the vector *x*.

Usage:

```
flag = N_VBufUnpack(x, buf)
```

6.3 NVECTOR functions used by KINSOL

In Table 6.2 below, we list the vector functions used in the `N_Vector` module used by 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 integrator module, while the remaining columns show function usage within the KINLS linear solvers interface, and the KINBBDPRE preconditioner module.

At this point, we should emphasize that the KINSOL user does not need to know anything about the usage of vector functions by the KINSOL code modules in order to use KINSOL. The information is presented as an implementation detail for the interested reader.

Table 6.2: List of vector functions usage by KINSOL code modules

Function name	KINSOL	KINLS	KINBBDPRE
N_VGetVectorID()			
N_VGetLength()	4		
N_VClone()	x	x	
N_VCloneEmpty()			
N_VDestroy()	x	x	
N_VSpace()	x	2	
N_VGetArrayPointer()	1	x	
N_VSetArrayPointer()	1		
N_VLinearSum()	x	x	
N_VConst()	x		
N_VProd()	x	x	
N_VDiv()	x		
N_VScale()	x	x	x
N_VAbs()	x		
N_VInv()	x		
N_VDotProd()	x	x	
N_VMaxNorm()	x		
N_VMin()	x		
N_VWL2Norm()	x	x	
N_VL1Norm()	3		
N_VConstrMask()	x		
N_VMinQuotient()	x		
N_VLinearCombination()	x	x	
N_VDotProdMulti()	x		

Special cases (numbers match markings in table):

1. These routines are only required if an internal difference-quotient routine for constructing *SUNMATRIX_DENSE* or *SUNMATRIX_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 `SUNLinearSolver` module that does not support the *SUNLinSolSetScalingVectors()* routine is supplied to KINSOL.

Each `SUNLinearSolver` object may require additional `N_Vector` routines not listed in the table above. Please see the the relevant descriptions of these modules in §8 for additional detail on their `N_Vector` requirements.

The vector functions listed in §6.2 that are *not* used by KINSOL are *N_VAddConst()*, *N_VWrmsNorm()*, *N_VWrmsNormMask()*, *N_VCompare()*, *N_VInvTest()*, and *N_VGetCommunicator()*. Therefore a user-supplied `N_Vector` module for KINSOL could omit these functions.

The optional function *N_VLinearCombination()* is only used when Anderson acceleration is enabled or the SPBCG, 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 §6.2.2 and §6.2.3 are unused and a user-supplied `N_Vector` module for KINSOL could omit these operations.

6.4 The NVECTOR_SERIAL Module

The serial implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_SERIAL, defines the *content* field of an `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 be included 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.

6.4.1 NVECTOR_SERIAL accessor macros

The following five 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(v)
This macro gives access to the contents of the serial vector `N_Vector` *v*.

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(v)
Access the *own_data* component of the serial `N_Vector` *v*.

Implementation:

```
#define NV_OWN_DATA_S(v) ( NV_CONTENT_S(v)->own_data )
```

NV_DATA_S(v)
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`.

Similarly, the assignment `NV_DATA_S(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

Implementation:

```
#define NV_DATA_S(v) ( NV_CONTENT_S(v)->data )
```

NV_LENGTH_S(v)
Access the *length* component of the serial `N_Vector` *v*.

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_LENGTH_S(v) ( NV_CONTENT_S(v)->length )
```

NV_Ith_S(v, i)
This macro gives access to the individual components of the *data* array of an `N_Vector`, using standard 0-based C indexing.

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] )
```

6.4.2 NVECTOR_SERIAL functions

The NVECTOR_SERIAL module defines serial implementations of all vector operations listed in §6.2.1, §6.2.2, §6.2.3, and §6.2.4. Their names are obtained from those in those sections by appending the suffix `_Serial` (e.g. `N_VDestroy_Serial`). All the standard vector operations listed in §6.2.1 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_Vector **N_VNew_Serial**(*sunindextype* vec_length, *SUNContext* sunctx)
This function creates and allocates memory for a serial `N_Vector`. Its only argument is the vector length.

N_Vector **N_VNewEmpty_Serial**(*sunindextype* vec_length, *SUNContext* sunctx)
This function creates a new serial `N_Vector` with an empty (`NULL`) data array.

N_Vector **N_VMake_Serial**(*sunindextype* vec_length, *realtype* *v_data, *SUNContext* sunctx)
This function creates and allocates memory for a serial vector with user-provided data array, *v_data*.

(This function does *not* allocate memory for `v_data` itself.)

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

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

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 that 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.

int **N_VEnableFusedOps_Serial**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableLinearCombination_Serial**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableScaleAddMulti_Serial**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableDotProdMulti_Serial**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearSumVectorArray_Serial**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleVectorArray_Serial**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableConstVectorArray_Serial**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableWrmsNormVectorArray_Serial**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableWrmsNormMaskVectorArray_Serial**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleAddMultiVectorArray_Serial**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearCombinationVectorArray_Serial**(*N_Vector* v, *booleantype* tf)
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), or equivalently v_data = N_VGetArrayPointer(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* to SUNFALSE. The functions 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 length.

6.4.3 NVECTOR_SERIAL Fortran Interface

The NVECTOR_SERIAL module provides a Fortran 2003 module for use from Fortran applications.

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 §10. We note that the module is accessible from the Fortran 2003 SUNDIALS integrators *without* separately linking to the `libsundials_fnvectorserial_mod` library.

6.5 The NVECTOR_PARALLEL Module

The NVECTOR_PARALLEL implementation of the NVECTOR module provided with SUNDIALS is based on MPI. It defines the *content* field of an `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, an 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 be included 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.

6.5.1 NVECTOR_PARALLEL accessor macros

The following seven 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(v)
: This macro gives access to the contents of the parallel `N_Vector` *v*.

  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(v)
: Access the *own_data* component of the parallel `N_Vector` *v*.

  Implementation:

  ```
  #define NV_OWN_DATA_P(v)    ( NV_CONTENT_P(v)->own_data )
  ```

NV_DATA_P(v)

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` into *data*.

Implementation:

```
#define NV_DATA_P(v)        ( NV_CONTENT_P(v)->data )
```

NV_LOCLENGTH_P(v)

The assignment `v_llen = NV_LOCLENGTH_P(v)` sets `v_llen` to be the length of the local part of `v`.

The call `NV_LOCLENGTH_P(v) = llen_v` sets the *local_length* of `v` to be `llen_v`.

Implementation:

```
#define NV_LOCLENGTH_P(v)   ( NV_CONTENT_P(v)->local_length )
```

NV_GLOBLENGTH_P(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_GLOBLENGTH_P(v) ( NV_CONTENT_P(v)->global_length )
```

NV_COMM_P(v)

This macro provides access to the MPI communicator used by the parallel `N_Vector` *v*.

Implementation:

```
#define NV_COMM_P(v) ( NV_CONTENT_P(v)->comm )
```

NV_Ith_P(v, i)

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] )
```

6.5.2 NVECTOR_PARALLEL functions

The NVECTOR_PARALLEL module defines parallel implementations of all vector operations listed in §6.2. Their names are obtained from the generic names by appending the suffix `_Parallel` (e.g. `N_VDestroy_Parallel`). The module NVECTOR_PARALLEL provides the following additional user-callable routines:

N_Vector **N_VNew_Parallel**(MPI_Comm comm, *sunindextype* local_length, *sunindextype* global_length, *SUNContext* sunctx)

This function creates and allocates memory for a parallel vector having global length *global_length*, having processor-local length *local_length*, and using the MPI communicator *comm*.

N_Vector **N_VNewEmpty_Parallel**(MPI_Comm comm, *sunindextype* local_length, *sunindextype* global_length, *SUNContext* sunctx)
This function creates a new parallel `N_Vector` with an empty (`NULL`) data array.

N_Vector **N_VMake_Parallel**(MPI_Comm comm, *sunindextype* local_length, *sunindextype* global_length, *realtype* *v_data, *SUNContext* sunctx)
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.)

sunindextype **N_VGetLocalLength_Parallel**(*N_Vector* v)
This function returns the local vector length.

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

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

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()*. This guarantees that 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.

int **N_VEnableFusedOps_Parallel**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableLinearCombination_Parallel**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableScaleAddMulti_Parallel**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableDotProdMulti_Parallel**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableLinearSumVectorArray_Parallel**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableScaleVectorArray_Parallel**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableConstVectorArray_Parallel**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableWrmsNormVectorArray_Parallel**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableWrmsNormMaskVectorArray_Parallel**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleAddMultiVectorArray_Parallel**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearCombinationVectorArray_Parallel**(*N_Vector* v, *booleantype* tf)
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 = N_VGetArrayPointer(v), or equivalently 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* to SUNFALSE. The routines 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.

6.5.3 NVECTOR_PARALLEL Fortran Interface

The NVECTOR_PARALLEL module provides a Fortran 2003 module for use from Fortran applications.

The fnvector_parallel_mod Fortran module defines interfaces to all NVECTOR_PARALLEL 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_Parallel is interfaced as FN_VNew_Parallel.

The Fortran 2003 NVECTOR_PARALLEL interface module can be accessed with the use statement, i.e. use fnvector_parallel_mod, and linking to the library libsundials_fnvectorparallel_mod.lib in addition to the C library. For details on where the library and module file fnvector_parallel_mod.mod are installed see §10. We note that the module is accessible from the Fortran 2003 SUNDIALS integrators *without* separately linking to the libsundials_fnvectorparallel_mod library.

6.6 The NVECTOR_OPENMP Module

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, the number of threads used is based on the supplied argument in the vector constructor.

```
struct _N_VectorContent_OpenMP {
  sunindextype length;
  booleantype own_data;
  realtype *data;
  int num_threads;
};
```

The header file to be included 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`.

6.6.1 NVECTOR_OPENMP accessor macros

The following six 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(v)
This macro gives access to the contents of the OpenMP vector `N_Vector` *v*.

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(v)
Access the *own_data* component of the OpenMP `N_Vector` *v*.

Implementation:

```
#define NV_OWN_DATA_OMP(v) ( NV_CONTENT_OMP(v)->own_data )
```

NV_DATA_OMP(v)
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`.

Similarly, the assignment `NV_DATA_OMP(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

Implementation:

```
#define NV_DATA_OMP(v) ( NV_CONTENT_OMP(v)->data )
```

NV_LENGTH_OMP(v)
Access the *length* component of the OpenMP `N_Vector` *v*.

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`.

Implementation:

```
#define NV_LENGTH_OMP(v) ( NV_CONTENT_OMP(v)->length )
```

NV_NUM_THREADS_OMP(v)
Access the *num_threads* component of the OpenMP `N_Vector` *v*.

The assignment `v_threads = NV_NUM_THREADS_OMP(v)` sets `v_threads` to be the *num_threads* of `v`. On the other hand, the call `NV_NUM_THREADS_OMP(v) = num_threads_v` sets the *num_threads* of `v` to be `num_threads_v`.

Implementation:

```
#define NV_NUM_THREADS_OMP(v) ( NV_CONTENT_OMP(v)->num_threads )
```

NV_Ith_OMP(v, i)
This macro gives access to the individual components of the *data* array of an `N_Vector`, using standard 0-based C indexing.

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] )
```

6.6.2 NVECTOR_OPENMP functions

The NVECTOR_OPENMP module defines OpenMP implementations of all vector operations listed in §6.2, §6.2.2, §6.2.3, and §6.2.4. Their names are obtained from those in those sections by appending the suffix `_OpenMP` (e.g. `N_VDestroy_OpenMP`). All the standard vector operations listed in §6.2 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_Vector **N_VNew_OpenMP**(*sunindextype* vec_length, int num_threads, *SUNContext* sunctx)
This function creates and allocates memory for a OpenMP `N_Vector`. Arguments are the vector length and number of threads.

N_Vector **N_VNewEmpty_OpenMP**(*sunindextype* vec_length, int num_threads, *SUNContext* sunctx)
This function creates a new OpenMP `N_Vector` with an empty (`NULL`) data array.

N_Vector **N_VMake_OpenMP**(*sunindextype* vec_length, *realtype* *v_data, int num_threads, *SUNContext* sunctx)
This function creates and allocates memory for a OpenMP vector with user-provided data array, *v_data*.

(This function does *not* allocate memory for `v_data` itself.)

void **N_VPrint_OpenMP**(*N_Vector* v)
This function prints the content of an OpenMP vector to `stdout`.

void **N_VPrintFile_OpenMP**(*N_Vector* v, FILE *outfile)
This function prints the content of an OpenMP vector to `outfile`.

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.

int **N_VEnableFusedOps_OpenMP**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableLinearCombination_OpenMP**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleAddMulti_OpenMP**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableDotProdMulti_OpenMP**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearSumVectorArray_OpenMP**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleVectorArray_OpenMP**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableConstVectorArray_OpenMP**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableWrmsNormVectorArray_OpenMP**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableWrmsNormMaskVectorArray_OpenMP**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleAddMultiVectorArray_OpenMP**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearCombinationVectorArray_OpenMP**(*N_Vector* v, *booleantype* tf)
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 = N_VGetArrayPointer(v), or equivalently 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* to SUNFALSE. The functions 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.

6.6.3 NVECTOR_OPENMP Fortran Interface

The NVECTOR_OPENMP module provides a Fortran 2003 module for use from Fortran applications.

The `fnvector_openmp_mod` Fortran module defines interfaces to all 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 §10.

6.7 The NVECTOR_PTHREADS Module

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 be included 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.

6.7.1 NVECTOR_PTHREADS accessor macros

The following six 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(v)
This macro gives access to the contents of the Pthreads vector `N_Vector` *v*.

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(v)
Access the *own_data* component of the Pthreads `N_Vector` *v*.

Implementation:

```
#define NV_OWN_DATA_PT(v) ( NV_CONTENT_PT(v)->own_data )
```

NV_DATA_PT(v)
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`.

Similarly, the assignment `NV_DATA_PT(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

Implementation:

```
#define NV_DATA_PT(v) ( NV_CONTENT_PT(v)->data )
```

NV_LENGTH_PT(v)
Access the *length* component of the Pthreads `N_Vector` *v*.

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`.

Implementation:

```
#define NV_LENGTH_PT(v) ( NV_CONTENT_PT(v)->length )
```

NV_NUM_THREADS_PT(v)
Access the *num_threads* component of the Pthreads `N_Vector` *v*.

The assignment `v_threads = NV_NUM_THREADS_PT(v)` sets `v_threads` to be the *num_threads* of `v`. On the other hand, the call `NV_NUM_THREADS_PT(v) = num_threads_v` sets the *num_threads* of `v` to be `num_threads_v`.

Implementation:

```
#define NV_NUM_THREADS_PT(v) ( NV_CONTENT_PT(v)->num_threads )
```

NV_Ith_PT(v, i)
This macro gives access to the individual components of the *data* array of an `N_Vector`, using standard 0-based C indexing.

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] )
```

6.7.2 NVECTOR_PTHREADS functions

The NVECTOR_PTHREADS module defines Pthreads implementations of all vector operations listed in §6.2, §6.2.2, §6.2.3, and §6.2.4. Their names are obtained from those in those sections by appending the suffix `_Pthreads` (e.g. N_VDestroy_Pthreads). All the standard vector operations listed in §6.2 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_Vector **N_VNew_Pthreads**(*sunindextype* vec_length, int num_threads, *SUNContext* sunctx)
This function creates and allocates memory for a Pthreads `N_Vector`. Arguments are the vector length and number of threads.

N_Vector **N_VNewEmpty_Pthreads**(*sunindextype* vec_length, int num_threads, *SUNContext* sunctx)
This function creates a new Pthreads `N_Vector` with an empty (`NULL`) data array.

N_Vector **N_VMake_Pthreads**(*sunindextype* vec_length, *realtype* *v_data, int num_threads, *SUNContext* sunctx)
This function creates and allocates memory for a Pthreads vector with user-provided data array, *v_data*.

(This function does *not* allocate memory for `v_data` itself.)

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

void **N_VPrintFile_Pthreads**(*N_Vector* v, FILE *outfile)
This function prints the content of a Pthreads vector to `outfile`.

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.

int **N_VEnableFusedOps_Pthreads**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableLinearCombination_Pthreads**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableScaleAddMulti_Pthreads**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableDotProdMulti_Pthreads**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableLinearSumVectorArray_Pthreads**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableScaleVectorArray_Pthreads**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableConstVectorArray_Pthreads**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableWrmsNormVectorArray_Pthreads**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableWrmsNormMaskVectorArray_Pthreads**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableScaleAddMultiVectorArray_Pthreads**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearCombinationVectorArray_Pthreads**(*N_Vector* v, *booleantype* tf)
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 = N_VGetArrayPointer(v), or equivalently v_data = NV_DATA_PT(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_Pthreads()*, *N_VMake_Pthreads()*, and N_VCloneVectorArrayEmpty_Pthreads() set the field *own_data* to SUNFALSE. The functions 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.

6.7.3 NVECTOR_PTHREADS Fortran Interface

The NVECTOR_PTHREADS module provides a Fortran 2003 module for use from Fortran applications.

The fnvector_pthreads_mod Fortran module defines interfaces to all 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 §10.

6.8 The NVECTOR_PARHYP Module

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_data;
  booleantype own_parvector;
  realtype *data;
  MPI_Comm comm;
  hypre_ParVector *x;
};
```

The header file to be included 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.

6.8.1 NVECTOR_PARHYP functions

The NVECTOR_PARHYP module defines implementations of all vector operations listed in §6.2 except for *N_VSetArrayPointer()* and *N_VGetArrayPointer()* because accessing raw vector data is handled by low-level HYPRE functions. As such, this vector is not available for use with SUNDIALS Fortran interfaces. When access to raw vector data is needed, one should extract the HYPRE vector first, and then use HYPRE methods to access the data. Usage examples of NVECTOR_PARHYP are provided in the `cvAdvDiff_non_ph.c` example programs for CVODE and the `ark_diurnal_kry_ph.c` example program for ARKODE.

The names of parhyp methods are obtained from those in §6.2, §6.2.2, §6.2.3, and §6.2.4 by appending the suffix `_ParHyp` (e.g. `N_VDestroy_ParHyp`). The module NVECTOR_PARHYP provides the following additional user-callable routines:

N_Vector **N_VNewEmpty_ParHyp**(MPI_Comm comm, *sunindextype* local_length, *sunindextype* global_length, *SUNContext* sunctx)
: This function creates a new parhyp `N_Vector` with the pointer to the HYPRE vector set to `NULL`.

N_Vector **N_VMake_ParHyp**(hypre_ParVector *x, *SUNContext* sunctx)
: This function creates an `N_Vector` wrapper around an existing HYPRE parallel vector. It does *not* allocate memory for `x` itself.

hypre_ParVector ***N_VGetVector_ParHyp**(*N_Vector* v)
: This function returns a pointer to the underlying HYPRE vector.

void **N_VPrint_ParHyp**(*N_Vector* v)
: This function prints the local content of a parhyp vector to `stdout`.

void **N_VPrintFile_ParHyp**(*N_Vector* v, FILE *outfile)
: 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.

int **N_VEnableFusedOps_ParHyp**(*N_Vector* v, *booleantype* tf)
: 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`.

int **N_VEnableLinearCombination_ParHyp**(*N_Vector* v, *booleantype* tf)
: 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`.

int **N_VEnableScaleAddMulti_ParHyp**(*N_Vector* v, *booleantype* tf)
: 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`.

int **N_VEnableDotProdMulti_ParHyp**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearSumVectorArray_ParHyp**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleVectorArray_ParHyp**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableConstVectorArray_ParHyp**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableWrmsNormVectorArray_ParHyp**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableWrmsNormMaskVectorArray_ParHyp**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleAddMultiVectorArray_ParHyp**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearCombinationVectorArray_ParHyp**(*N_Vector* v, *booleantype* tf)
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. The functions N_VDestroy_ParHyp() and N_VDestroyVectorArray_ParHyp() will not attempt to delete an underlying HYPRE vector for any N_Vector with *own_parvector* set to SUNFALSE. In such a case, it is the user's responsibility to delete the underlying vector.
- To maximize efficiency, vector operations in the NVECTOR_PARHYP implementation that have more than one N_Vector argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

6.9 The NVECTOR_PETSC Module

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 be included 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.

6.9.1 NVECTOR_PETSC functions

The NVECTOR_PETSC module defines implementations of all vector operations listed in §6.2 except for *N_VGetArrayPointer()* and *N_VSetArrayPointer()*. As such, this vector cannot be used with SUNDIALS Fortran interfaces. When access to raw vector data is needed, it is recommended to extract the PETSc vector first, and then use PETSc methods to access the data. Usage examples of NVECTOR_PETSC is provided in example programs for IDA.

The names of vector operations are obtained from those in §6.2, §6.2.2, §6.2.3, and §6.2.4 by appending the suffice `_Petsc` (e.g. `N_VDestroy_Petsc`). The module NVECTOR_PETSC provides the following additional user-callable routines:

N_Vector **N_VNewEmpty_Petsc**(MPI_Comm comm, *sunindextype* local_length, *sunindextype* global_length, *SUNContext* sunctx)
: This function creates a new PETSC `N_Vector` with the pointer to the wrapped PETSc vector set to `NULL`. It is used by the `N_VMake_Petsc` and `N_VClone_Petsc` implementations. It should be used only with great caution.

N_Vector **N_VMake_Petsc**(Vec *pvec, *SUNContext* sunctx)
: This function creates and allocates memory for an NVECTOR_PETSC wrapper with a user-provided PETSc vector. It does *not* allocate memory for the vector `pvec` itself.

Vec ***N_VGetVector_Petsc**(*N_Vector* v)
: This function returns a pointer to the underlying PETSc vector.

void **N_VPrint_Petsc**(*N_Vector* v)
: This function prints the global content of a wrapped PETSc vector to `stdout`.

void **N_VPrintFile_Petsc**(*N_Vector* v, const char fname[])
: 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.

int **N_VEnableFusedOps_Petsc**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearCombination_Petsc**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleAddMulti_Petsc**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableDotProdMulti_Petsc**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearSumVectorArray_Petsc**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleVectorArray_Petsc**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableConstVectorArray_Petsc**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableWrmsNormVectorArray_Petsc**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableWrmsNormMaskVectorArray_Petsc**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleAddMultiVectorArray_Petsc**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearCombinationVectorArray_Petsc**(*N_Vector* v, *booleantype* tf)
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 recommeded to extract the PETSc vector via x_vec = N_VGetVector_Petsc(v); and then access components using appropriate PETSc functions.
- The functions *N_VNewEmpty_Petsc()*, *N_VMake_Petsc()*, and N_VCloneVectorArrayEmpty_Petsc() set the field *own_data* to SUNFALSE. The routines N_VDestroy_Petsc() and N_VDestroyVectorArray_Petsc() will not attempt to free the pointer pvec for any N_Vector with *own_data* set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the pvec pointer.
- To maximize efficiency, vector operations in the NVECTOR_PETSC implementation that have more than one N_Vector argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

6.10 The NVECTOR_CUDA Module

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_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), boolean flags that indicate if the vector owns the execution policies and memory helper objects (i.e., it is in change 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, and a private data structure which holds additonal members that should not be accessed directly.

When instantiated with *N_VNew_Cuda()*, 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_Cuda()* constructor. To use CUDA managed memory, the constructors *N_VNewManaged_Cuda()* and *N_VMakeManaged_Cuda()* are provided. Additionally, a user-defined `SUNMemoryHelper` for allocating/freeing data can be provided with the constructor *N_VNewWithMemHelp_Cuda()*. Details on each of these constructors are provided below.

To use the NVECTOR_CUDA module, include `nvector_cuda.h` and link to the library `libsundials_nveccuda.lib`. The extension, `.lib`, is typically `.so` for shared libraries and `.a` for static libraries.

6.10.1 NVECTOR_CUDA functions

Unlike other native SUNDIALS vector types, the NVECTOR_CUDA module does not provide macros to access its member variables. Instead, user should use the accessor functions:

realtype ***N_VGetHostArrayPointer_Cuda**(*N_Vector* v)
This function returns pointer to the vector data on the host.

realtype ***N_VGetDeviceArrayPointer_Cuda**(*N_Vector* v)
This function returns pointer to the vector data on the device.

booleantype **N_VIsManagedMemory_Cuda**(*N_Vector* v)
This function returns a boolean flag indiciating if the vector data array is in managed memory or not.

The NVECTOR_CUDA module defines implementations of all standard vector operations defined in §6.2, §6.2.2, §6.2.3, and §6.2.4, except for *N_VSetArrayPointer()*, and, if using unmanaged memory, *N_VGetArrayPointer()*. As such, this vector can only be used with 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 example programs for CVODE [21].

The names of vector operations are obtained from those in §6.2, §6.2.2, §6.2.3, and §6.2.4 by appending the suffix _Cuda (e.g. N_VDestroy_Cuda). The module NVECTOR_CUDA provides the following additional user-callable routines:

N_Vector **N_VNew_Cuda**(*sunindextype* length, *SUNContext* sunctx)
This function creates and allocates memory for a CUDA N_Vector. The vector data array is allocated on both the host and device.

N_Vector **N_VNewManaged_Cuda**(*sunindextype* vec_length, *SUNContext* sunctx)
This function creates and allocates memory for a CUDA N_Vector. The vector data array is allocated in managed memory.

N_Vector **N_VNewWithMemHelp_Cuda**(*sunindextype* length, *booleantype* use_managed_mem, *SUNMemoryHelper* helper, *SUNContext* sunctx)
This function creates a new CUDA N_Vector with a user-supplied SUNMemoryHelper for allocating/freeing memory.

N_Vector **N_VNewEmpty_Cuda**(*sunindextype* vec_length, *SUNContext* sunctx)
This function creates a new CUDA N_Vector 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.

N_Vector **N_VMake_Cuda**(*sunindextype* vec_length, *realtype* *h_vdata, *realtype* *d_vdata, *SUNContext* sunctx)
This function creates a CUDA N_Vector with user-supplied vector data arrays for the host and the device.

N_Vector **N_VMakeManaged_Cuda**(*sunindextype* vec_length, *realtype* *vdata, *SUNContext* sunctx)
This function creates a CUDA N_Vector with a user-supplied managed memory data array.

N_Vector **N_VMakeWithManagedAllocator_Cuda**(*sunindextype* length, void *(*allocfn)(size_t size), void (*freefn)(void *ptr))
This function creates a CUDA N_Vector 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.

The module NVECTOR_CUDA also provides the following user-callable routines:

void **N_VSetKernelExecPolicy_Cuda**(*N_Vector* v, SUNCudaExecPolicy *stream_exec_policy, SUNCudaExecPolicy *reduce_exec_policy)
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 *SUNCudaBlockReduceAtomicExecPolicy()*. 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 §6.10.2 below for more information about the *SUNCudaExecPolicy* class. Providing NULL for an argument will result in the default policy being restored.

Note: 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

---

realtype ***N_VCopyToDevice_Cuda**(*N_Vector* v)
This function copies host vector data to the device.

realtype ***N_VCopyFromDevice_Cuda**(*N_Vector* v)
This function copies vector data from the device to the host.

void **N_VPrint_Cuda**(*N_Vector* v)
This function prints the content of a CUDA vector to stdout.

void **N_VPrintFile_Cuda**(*N_Vector* v, FILE *outfile)
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.

int **N_VEnableFusedOps_Cuda**(*N_Vector* v, *booleantype* tf)
: 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.

int **N_VEnableLinearCombination_Cuda**(*N_Vector* v, *booleantype* tf)
: 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.

int **N_VEnableScaleAddMulti_Cuda**(*N_Vector* v, *booleantype* tf)
: 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.

int **N_VEnableDotProdMulti_Cuda**(*N_Vector* v, *booleantype* tf)
: 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.

int **N_VEnableLinearSumVectorArray_Cuda**(*N_Vector* v, *booleantype* tf)
: 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.

int **N_VEnableScaleVectorArray_Cuda**(*N_Vector* v, *booleantype* tf)
: 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.

int **N_VEnableConstVectorArray_Cuda**(*N_Vector* v, *booleantype* tf)
: 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.

int **N_VEnableWrmsNormVectorArray_Cuda**(*N_Vector* v, *booleantype* tf)
: 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.

int **N_VEnableWrmsNormMaskVectorArray_Cuda**(*N_Vector* v, *booleantype* tf)
: 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.

int **N_VEnableScaleAddMultiVectorArray_Cuda**(*N_Vector* v, *booleantype* tf)
: 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.

int **N_VEnableLinearCombinationVectorArray_Cuda**(*N_Vector* v, *booleantype* tf)
: 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 recommeded to use functions *N_VGetDeviceArrayPointer_Cuda()* or *N_VGetHostArrayPointer_Cuda()*. However, when using managed memory, the function *N_VGetArrayPointer()* may also be used.

- To maximize efficiency, vector operations in the NVECTOR_CUDA implementation that have more than one `N_Vector` argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

6.10.2 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::cuda::ExecPolicy` 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 `SUNCudaExecPolicy` class is defined as

typedef sundials::cuda::ExecPolicy **SUNCudaExecPolicy**

where the `sundials::cuda::ExecPolicy` class is defined in the header file `sundials_cuda_policies.hpp`, as follows:

```
class ExecPolicy
{
public:
   ExecPolicy(cudaStream_t stream = 0) : stream_(stream) { }
   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 const cudaStream_t* stream() const { return (&stream_); }
   virtual ExecPolicy* clone() const = 0;
   ExecPolicy* clone_new_stream(cudaStream_t stream) const {
      ExecPolicy* ex = clone();
      ex->stream_ = stream;
      return ex;
   }
   virtual bool atomic() const { return false; }
   virtual ~ExecPolicy() {}
protected:
   cudaStream_t stream_;
};
```

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::cuda::ThreadDirectExecPolicy` (aka in the global namespace as `SUNCudaThreadDirectExecPolicy`) class is a good example of a what a custom execution policy may look like:

```
class ThreadDirectExecPolicy : public ExecPolicy
{
public:
   ThreadDirectExecPolicy(const size_t blockDim, cudaStream_t stream = 0)
      : blockDim_(blockDim), ExecPolicy(stream)
   {}

   ThreadDirectExecPolicy(const ThreadDirectExecPolicy& ex)
      : blockDim_(ex.blockDim_), ExecPolicy(ex.stream_)
   {}

   virtual size_t gridSize(size_t numWorkUnits = 0, size_t /*blockDim*/ = 0) const
   {
```

(continues on next page)

(continued from previous page)

```
      /* ceil(n/m) = floor((n + m - 1) / m) */
      return (numWorkUnits + blockSize() - 1) / blockSize();
   }

   virtual size_t blockSize(size_t /*numWorkUnits*/ = 0, size_t /*gridDim*/ = 0) const
   {
      return blockDim_;
   }

   virtual ExecPolicy* clone() const
   {
      return static_cast<ExecPolicy*>(new ThreadDirectExecPolicy(*this));
   }

private:
   const size_t blockDim_;
};
```

In total, SUNDIALS provides 3 execution policies:

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.

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.

SUNCudaBlockReduceExecPolicy(const size_t blockDim, const cudaStream_t stream = 0)
: Is for kernels performing a reduction across indvidual 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.

SUNCudaBlockReduceAtomicExecPolicy(const size_t blockDim, const cudaStream_t stream = 0)
: Is for kernels performing a reduction across indvidual thread blocks using atomic operations. 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 (e.g. a *SUNMatrix* and an *N_Vector*) since they do not hold any modifiable state information.

6.11 The NVECTOR_HIP Module

The NVECTOR_HIP module is an NVECTOR implementation using the AMD ROCm HIP library [35]. 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_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 additonal members that should not be accessed directly.

When instantiated with *N_VNew_Hip()*, 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_Hip()* constructor. To use managed memory, the constructors *N_VNewManaged_Hip()* and *N_VMakeManaged_Hip()* are provided. Additionally, a user-defined `SUNMemoryHelper` for allocating/freeing data can be provided with the constructor *N_VNewWithMemHelp_Hip()*. Details on each of these constructors are provided below.

To use the NVECTOR_HIP module, include `nvector_hip.h` and link to the library `libsundials_nvechip.lib`. The extension, `.lib`, is typically `.so` for shared libraries and `.a` for static libraries.

6.11.1 NVECTOR_HIP functions

Unlike other native SUNDIALS vector types, the NVECTOR_HIP module does not provide macros to access its member variables. Instead, user should use the accessor functions:

realtype ***N_VGetHostArrayPointer_Hip**(*N_Vector* v)
This function returns pointer to the vector data on the host.

realtype ***N_VGetDeviceArrayPointer_Hip**(*N_Vector* v)
This function returns pointer to the vector data on the device.

booleantype **N_VIsManagedMemory_Hip**(*N_Vector* v)
This function returns a boolean flag indiciating if the vector data array is in managed memory or not.

The NVECTOR_HIP module defines implementations of all standard vector operations defined in §6.2, §6.2.2, §6.2.3, and §6.2.4, except for *N_VSetArrayPointer()*. The names of vector operations are obtained from those in §6.2, §6.2.2, §6.2.3, and §6.2.4 by appending the suffix `_Hip` (e.g. `N_VDestroy_Hip()`). The module NVECTOR_HIP provides the following additional user-callable routines:

N_Vector **N_VNew_Hip**(*sunindextype* length, *SUNContext* sunctx)
This function creates and allocates memory for a HIP `N_Vector`. The vector data array is allocated on both the host and device.

N_Vector **N_VNewManaged_Hip**(*sunindextype* vec_length, *SUNContext* sunctx)
This function creates and allocates memory for a HIP `N_Vector`. The vector data array is allocated in managed memory.

N_Vector **N_VNewWithMemHelp_Hip**(*sunindextype* length, *booleantype* use_managed_mem, *SUNMemoryHelper* helper, *SUNContext* sunctx)
This function creates a new HIP `N_Vector` with a user-supplied SUNMemoryHelper for allocating/freeing memory.

N_Vector **N_VNewEmpty_Hip**(*sunindextype* vec_length, *SUNContext* sunctx)
This function creates a new HIP `N_Vector` 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.

N_Vector **N_VMake_Hip**(*sunindextype* vec_length, *realtype* *h_vdata, *realtype* *d_vdata, *SUNContext* sunctx)
This function creates a HIP `N_Vector` with user-supplied vector data arrays for the host and the device.

N_Vector **N_VMakeManaged_Hip**(*sunindextype* vec_length, *realtype* *vdata, *SUNContext* sunctx)
This function creates a HIP `N_Vector` with a user-supplied managed memory data array.

The module NVECTOR_HIP also provides the following user-callable routines:

void **N_VSetKernelExecPolicy_Hip**(*N_Vector* v, SUNHipExecPolicy *stream_exec_policy, SUNHipExecPolicy *reduce_exec_policy)
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 (32 for NVIDIA GPUs, 64 for AMD GPUs). See §6.11.2 below for more information about the *SUNHipExecPolicy* class. Providing `NULL` for an argument will result in the default policy being restored.

Note: 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*

---

realtype ***N_VCopyToDevice_Hip**(*N_Vector* v)
This function copies host vector data to the device.

realtype ***N_VCopyFromDevice_Hip**(*N_Vector* v)
This function copies vector data from the device to the host.

void **N_VPrint_Hip**(*N_Vector* v)
This function prints the content of a HIP vector to `stdout`.

void **N_VPrintFile_Hip**(*N_Vector* v, FILE *outfile)
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.

int **N_VEnableFusedOps_Hip**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearCombination_Hip**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleAddMulti_Hip**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableDotProdMulti_Hip**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearSumVectorArray_Hip**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleVectorArray_Hip**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableConstVectorArray_Hip**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableWrmsNormVectorArray_Hip**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableWrmsNormMaskVectorArray_Hip**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleAddMultiVectorArray_Hip**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearCombinationVectorArray_Hip**(*N_Vector* v, *booleantype* tf)
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 recommeded 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.

6.11.2 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::hip::ExecPolicy` 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 `SUNHipExecPolicy` class is defined as

typedef sundials::hip::ExecPolicy **SUNHipExecPolicy**

where the `sundials::hip::ExecPolicy` class is defined in the header file `sundials_hip_policies.hpp`, as follows:

```
class ExecPolicy
{
public:
   ExecPolicy(hipStream_t stream = 0) : stream_(stream) { }
   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 const hipStream_t* stream() const { return (&stream_); }
   virtual ExecPolicy* clone() const = 0;
   ExecPolicy* clone_new_stream(hipStream_t stream) const {
      ExecPolicy* ex = clone();
      ex->stream_ = stream;
      return ex;
   }
   virtual bool atomic() const { return false; }
   virtual ~ExecPolicy() {}
protected:
   hipStream_t stream_;
};
```

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::hip::ThreadDirectExecPolicy` (aka in the global namespace as `SUNHipThreadDirectExecPolicy`) class is a good example of a what a custom execution policy may look like:

```
class ThreadDirectExecPolicy : public ExecPolicy
{
public:
   ThreadDirectExecPolicy(const size_t blockDim, hipStream_t stream = 0)
      : blockDim_(blockDim), ExecPolicy(stream)
   {}

   ThreadDirectExecPolicy(const ThreadDirectExecPolicy& ex)
      : blockDim_(ex.blockDim_), ExecPolicy(ex.stream_)
   {}

   virtual size_t gridSize(size_t numWorkUnits = 0, size_t /*blockDim*/ = 0) const
   {
      /* ceil(n/m) = floor((n + m - 1) / m) */
      return (numWorkUnits + blockSize() - 1) / blockSize();
   }

   virtual size_t blockSize(size_t /*numWorkUnits*/ = 0, size_t /*gridDim*/ = 0) const
   {
```

(continues on next page)

(continued from previous page)

```
      return blockDim_;
   }

   virtual ExecPolicy* clone() const
   {
      return static_cast<ExecPolicy*>(new ThreadDirectExecPolicy(*this));
   }

private:
   const size_t blockDim_;
};
```

In total, SUNDIALS provides 4 execution policies:

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.

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.

SUNHipBlockReduceExecPolicy(const size_t blockDim, const hipStream_t stream = 0)
: Is for kernels performing a reduction across indvidual 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.

SUNHipBlockReduceAtomicExecPolicy(const size_t blockDim, const hipStream_t stream = 0)
: Is for kernels performing a reduction across indvidual thread blocks using atomic operations. 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 (e.g. a *SUNMatrix* and an *N_Vector*) since they do not hold any modifiable state information.

6.12 The NVECTOR_RAJA Module

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_data;
   realtype*    host_data;
   realtype*    device_data;
   void*        priv; /* 'private' data */
};
```

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, 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 is `nvector_raja.h`. The installed module library to link to is `libsundials_nveccudaraja.lib` when using the CUDA backend, `libsundials_nvechipraja.lib` when using the HIP backend, and `libsundials_nvecsyclraja.lib` when using the SYCL backend. The extension `.lib` is typically `.so` for shared libraries `.a` for static libraries.

6.12.1 NVECTOR_RAJA functions

Unlike other native SUNDIALS vector types, the NVECTOR_RAJA module does not provide macros to access its member variables. Instead, user should use the accessor functions:

realtype ***N_VGetHostArrayPointer_Raja**(*N_Vector* v)
This function returns pointer to the vector data on the host.

realtype ***N_VGetDeviceArrayPointer_Raja**(*N_Vector* v)
This function returns pointer to the vector data on the device.

booleantype **N_VIsManagedMemory_Raja**(*N_Vector* v)
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 §6.2, §6.2.2, §6.2.3, and §6.2.4, except for *N_VDotProdMulti()*, *N_VWrmsNormVectorArray()*, and *N_VWrmsNormMaskVectorArray()* as support for arrays of reduction vectors is not yet supported in RAJA. These functions will be added to the NVECTOR_RAJA implementation in the future. Additionally, the operations *N_VGetArrayPointer()* and *N_VSetArrayPointer()* are not implemented by the RAJA vector. As such, this vector cannot be used with SUNDIALS direct

solvers and preconditioners. The NVECTOR_RAJA module provides separate functions to access data on the host and on the device. It also provides methods for copying from the host to the device and vice versa. Usage examples of NVECTOR_RAJA are provided in some example programs for CVODE [21].

The names of vector operations are obtained from those in §6.2, §6.2.2, §6.2.3, and §6.2.4 by appending the suffix `_Raja` (e.g. `N_VDestroy_Raja`). The module NVECTOR_RAJA provides the following additional user-callable routines:

N_Vector **N_VNew_Raja**(*sunindextype* vec_length, *SUNContext* sunctx)
This function creates and allocates memory for a RAJA `N_Vector`. The memory is allocated on both the host and the device. Its only argument is the vector length.

N_Vector **N_VNewManaged_Raja**(*sunindextype* vec_length, *SUNContext* sunctx)
This function creates and allocates memory for a RAJA `N_Vector`. The vector data array is allocated in managed memory.

N_Vector **N_VMake_Raja**(*sunindextype* length, *realtype* *h_data, *realtype* *v_data, *SUNContext* sunctx)
This function creates an NVECTOR_RAJA with user-supplied host and device data arrays. This function does not allocate memory for data itself.

N_Vector **N_VMakeManaged_Raja**(*sunindextype* length, *realtype* *vdata, *SUNContext* sunctx)
This function creates an NVECTOR_RAJA with a user-supplied managed memory data array. This function does not allocate memory for data itself.

N_Vector **N_VNewWithMemHelp_Raja**(*sunindextype* length, *booleantype* use_managed_mem, *SUNMemoryHelper* helper, *SUNContext* sunctx)
This function creates an NVECTOR_RAJA with a user-supplied SUNMemoryHelper for allocating/freeing memory.

N_Vector **N_VNewEmpty_Raja**()
This function creates a new `N_Vector` 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.

void **N_VCopyToDevice_Raja**(*N_Vector* v)
This function copies host vector data to the device.

void **N_VCopyFromDevice_Raja**(*N_Vector* v)
This function copies vector data from the device to the host.

void **N_VPrint_Raja**(*N_Vector* v)
This function prints the content of a RAJA vector to `stdout`.

void **N_VPrintFile_Raja**(*N_Vector* v, FILE *outfile)
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.

int **N_VEnableFusedOps_Raja**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableLinearCombination_Raja**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableScaleAddMulti_Raja**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearSumVectorArray_Raja**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleVectorArray_Raja**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableConstVectorArray_Raja**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleAddMultiVectorArray_Raja**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearCombinationVectorArray_Raja**(*N_Vector* v, *booleantype* tf)
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 NVECTOR_RAJA vector, 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.

6.13 The NVECTOR_SYCL Module

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;
```

(continues on next page)

(continued from previous page)

```
   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.

6.13.1 NVECTOR_SYCL functions

The NVECTOR_SYCL module implementations of all vector operations listed in §6.2, §6.2.2, §6.2.3, and §6.2.4, except for *N_VDotProdMulti()*, *N_VWrmsNormVectorArray()*, *N_VWrmsNormMaskVectorArray()* as support for arrays of reduction vectors is not yet supported. These functions 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_Vector **N_VNew_Sycl**(sunindextype vec_length, sycl::queue *Q, SUNContext sunctx)
: This function creates and allocates memory for an NVECTOR_SYCL. Vector data arrays are allocated on both the host and the device associated with the input queue. All operation are launched in the provided queue.

N_Vector **N_VNewManaged_Sycl**(sunindextype vec_length, sycl::queue *Q, SUNContext sunctx)
: This function creates and allocates memory for a NVECTOR_SYCL. The vector data array is allocated in managed (shared) memory using the input queue. All operation are launched in the provided queue.

N_Vector **N_VMake_Sycl**(sunindextype length, realtype *h_vdata, realtype *d_vdata, sycl::queue *Q, SUNContext sunctx)
: This function creates an NVECTOR_SYCL with user-supplied host and device data arrays. This function does not allocate memory for data itself. All operation are launched in the provided queue.

N_Vector **N_VMakeManaged_Sycl**(sunindextype length, realtype *vdata, sycl::queue *Q, SUNContext sunctx)
: This function creates an NVECTOR_SYCL with a user-supplied managed (shared) data array. This function does not allocate memory for data itself. All operation are launched in the provided queue.

N_Vector **N_VNewWithMemHelp_Sycl**(sunindextype length, booleantype use_managed_mem, SUNMemoryHelper helper, sycl::queue *Q, SUNContext sunctx)
: This function creates an NVECTOR_SYCL with a user-supplied SUNMemoryHelper for allocating/freeing memory. All operation are launched in the provided queue.

N_Vector **N_VNewEmpty_Sycl**()
This function creates a new N_Vector 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.

realtype ***N_VGetHostArrayPointer_Sycl**(N_Vector v)
This function returns a pointer to the vector host data array.

realtype ***N_VGetDeviceArrayPointer_Sycl**(N_Vector v)
This function returns a pointer to the vector device data array.

void **N_VSetHostArrayPointer_Sycl**(realtype *h_vdata, N_Vector v)
This function sets the host array pointer in the vector v.

void **N_VSetDeviceArrayPointer_Sycl**(realtype *d_vdata, N_Vector v)
This function sets the device array pointer in the vector v.

void **N_VCopyToDevice_Sycl**(N_Vector v)
This function copies host vector data to the device.

void **N_VCopyFromDevice_Sycl**(N_Vector v)
This function copies vector data from the device to the host.

booleantype **N_VIsManagedMemory_Sycl**(N_Vector v)
This function returns SUNTRUE if the vector data is allocated as managed (shared) memory otherwise it returns SUNFALSE.

The following user-callable function is provided to set the execution policies for how SYCL kernels are launched on a device.

int **N_VSetKernelExecPolicy_Sycl**(N_Vector v, *SUNSyclExecPolicy* *stream_exec_policy, *SUNSyclExecPolicy* *reduce_exec_policy)
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 §6.13.2 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.

void **N_VPrint_Sycl**(N_Vector v)
This function prints the host data array to stdout.

void **N_VPrintFile_Sycl**(N_Vector v, FILE *outfile)
This function prints the host data array 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 with *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 constructors above will have the default settings for the NVECTOR_SYCL module.

int **N_VEnableFusedOps_Sycl**(N_Vector v, booleantype tf)
: 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.

int **N_VEnableLinearCombination_Sycl**(N_Vector v, booleantype tf)
: 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.

int **N_VEnableScaleAddMulti_Sycl**(N_Vector v, booleantype tf)
: 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.

int **N_VEnableLinearSumVectorArray_Sycl**(N_Vector v, booleantype tf)
: 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.

int **N_VEnableScaleVectorArray_Sycl**(N_Vector v, booleantype tf)
: 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.

int **N_VEnableConstVectorArray_Sycl**(N_Vector v, booleantype tf)
: 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.

int **N_VEnableScaleAddMultiVectorArray_Sycl**(N_Vector v, booleantype tf)
: 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.

int **N_VEnableLinearCombinationVectorArray_Sycl**(N_Vector v, booleantype tf)
: 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 NVECTOR_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.

6.13.2 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::sycl::ExecPolicy` 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 `SUNSyclExecPolicy` class is defined as

typedef sundials::sycl::ExecPolicy **SUNSyclExecPolicy**

where the `sundials::sycl::ExecPolicy` class is defined in the header file `sundials_sycl_policies.hpp`, as follows:

```
class ExecPolicy
{
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 ExecPolicy* clone() const = 0;
   virtual ~ExecPolicy() {}
};
```

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::sycl::ThreadDirectExecPolicy` (aka in the global namespace as `SUNSyclThreadDirectExecPolicy`) class is a good example of a what a custom execution policy may look like:

```
class ThreadDirectExecPolicy : public ExecPolicy
{
public:
   ThreadDirectExecPolicy(const size_t blockDim)
      : blockDim_(blockDim)
   {}

   ThreadDirectExecPolicy(const ThreadDirectExecPolicy& 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 ExecPolicy* clone() const
```

(continues on next page)

(continued from previous page)

```
  {
    return static_cast<ExecPolicy*>(new ThreadDirectExecPolicy(*this));
  }

private:
  const size_t blockDim_;
};
```

SUNDIALS provides the following execution policies:

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).

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).

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, SUNContext sunctx);
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.

6.14 The NVECTOR_OPENMPDEV Module

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 contiguousdata 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.

6.14.1 NVECTOR_OPENMPDEV accessor macros

The following macros are provided to access the content of an NVECTOR_OPENMPDEV vector.

NV_CONTENT_OMPDEV(v)
This macro gives access to the contents of the NVECTOR_OPENMPDEV `N_Vector v`.

The assignment `v_cont = NV_CONTENT_S(v)` sets `v_cont` to be a pointer to the NVECTOR_OPENMPDEV content structure.

Implementation:

```
#define NV_CONTENT_OMPDEV(v) ( (N_VectorContent_OpenMPDEV)(v->content) )
```

NV_OWN_DATA_OMPDEV(v)
Access the *own_data* component of the OpenMPDEV `N_Vector v`.

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`.

Implementation:

```
#define NV_OWN_DATA_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->own_data )
```

NV_DATA_HOST_OMPDEV(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`.

Implementation:

```
#define NV_DATA_HOST_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->host_data )
```

NV_DATA_DEV_OMPDEV(v)
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`.

Implementation:

```
#define NV_DATA_DEV_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->dev_data )
```

NV_LENGTH_OMPDEV(V)
Access the *length* component of the OpenMPDEV `N_Vector v`.

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`.

```
#define NV_LENGTH_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->length )
```

6.14.2 NVECTOR_OPENMPDEV functions

The NVECTOR_OPENMPDEV module defines OpenMP device offloading implementations of all vector operations listed in §6.2, §6.2.2, §6.2.3, and §6.2.4, except for *N_VSetArrayPointer()*. As such, this vector cannot be used 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 the vector operations are obtained from those in §6.2, §6.2.2, §6.2.3, and §6.2.4 by appending the suffix `_OpenMPDEV` (e.g. `N_VDestroy_OpenMPDEV`). The module NVECTOR_OPENMPDEV provides the following additional user-callable routines:

N_Vector **N_VNew_OpenMPDEV**(*sunindextype* vec_length, *SUNContext* sunctx)
This function creates and allocates memory for an NVECTOR_OPENMPDEV `N_Vector`.

N_Vector **N_VNewEmpty_OpenMPDEV**(*sunindextype* vec_length, *SUNContext* sunctx)
This function creates a new NVECTOR_OPENMPDEV `N_Vector` with an empty (`NULL`) data array.

N_Vector **N_VMake_OpenMPDEV**(*sunindextype* vec_length, *realtype* *h_vdata, *realtype* *d_vdata, *SUNContext* sunctx)
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.

realtype ***N_VGetHostArrayPointer_OpenMPDEV**(*N_Vector* v)
This function returns a pointer to the host data array.

realtype ***N_VGetDeviceArrayPointer_OpenMPDEV**(*N_Vector* v)
This function returns a pointer to the device data array.

void **N_VPrint_OpenMPDEV**(*N_Vector* v)
This function prints the content of an NVECTOR_OPENMPDEV vector to `stdout`.

void **N_VPrintFile_OpenMPDEV**(*N_Vector* v, FILE *outfile)
This function prints the content of an NVECTOR_OPENMPDEV vector to `outfile`.

void **N_VCopyToDevice_OpenMPDEV**(*N_Vector* v)
This function copies the content of an NVECTOR_OPENMPDEV vector's host data array to the device data array.

void **N_VCopyFromDevice_OpenMPDEV**(*N_Vector* v)
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.

int **N_VEnableFusedOps_OpenMPDEV**(*N_Vector* v, *booleantype* tf)
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`.

int **N_VEnableLinearCombination_OpenMPDEV**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleAddMulti_OpenMPDEV**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableDotProdMulti_OpenMPDEV**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearSumVectorArray_OpenMPDEV**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleVectorArray_OpenMPDEV**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableConstVectorArray_OpenMPDEV**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableWrmsNormVectorArray_OpenMPDEV**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableWrmsNormMaskVectorArray_OpenMPDEV**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableScaleAddMultiVectorArray_OpenMPDEV**(*N_Vector* v, *booleantype* tf)
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.

int **N_VEnableLinearCombinationVectorArray_OpenMPDEV**(*N_Vector* v, *booleantype* tf)
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 = N_VGetArrayPointer(v) for the host array or v_data = N_VGetDeviceArrayPointer(v) for the device array, or equivalently to use the macros h_data = NV_DATA_HOST_OMPDEV(v) for the host array or v_data = NV_DATA_DEV_OMPDEV(v) for the device array, and then access h_data[i] or v_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* to SUNFALSE. The functions 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 pointers.
- 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 length.

6.15 The NVECTOR_TRILINOS Module

The NVECTOR_TRILINOS module is an NVECTOR wrapper around the Trilinos Tpetra vector. The interface to Tpetra is implemented in the sundials::trilinos::nvector_tpetra::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::trilinos::nvector_tpetra::TpetraVectorInterface class is defined as:

typedef Tpetra::Vector<realtype, int, sunindextype> **vector_type**;

The Tpetra vector will use the SUNDIALS-specified realtype as its scalar type, int as the local ordinal type, and sunindextype as the global ordinal type. This type definition will use Tpetra's default node type. Available Kokkos node types as of the 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.

6.15.1 NVECTOR_TRILINOS functions

The NVECTOR_TRILINOS module defines implementations of all vector operations listed in §6.2, §6.2.2, §6.2.3, and §6.2.4, except for *N_VGetArrayPointer()* and *N_VSetArrayPointer()*. As such, this vector cannot be used 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.

The names of vector operations are obtained from those in §6.2 by appending the suffice _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::trilinos::nvector_tpetra::TpetraVector, located in the file SundialsTpetraVectorKernels.hpp. The module NVECTOR_TRILINOS provides the following additional user-callable routines:

Teuchos::RCP<*vector_type*> **N_VGetVector_Trilinos**(N_Vector v)
: 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.

N_Vector **N_VMake_Trilinos**(Teuchos::RCP<*vector_type*> v)
: 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.

Notes

- The template parameter `vector_type` should be set as:

```
typedef sundials::trilinos::nvector_tpetra::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 recommeded 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.

6.16 The NVECTOR_MANYVECTOR Module

The NVECTOR_MANYVECTOR module is designed to facilitate problems with an inherent data partitioning within a computational node for the solution vector. 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. Two potential use cases for this flexibility include:

A. *Heterogenous computational architectures*: for data partitioning between different computing resources on a node, architecture-specific subvectors may be created for each partition. For example, a user could create one GPU-accelerated component based on *NVECTOR_CUDA*, and another CPU threaded component based on *NVECTOR_OPENMP*.

B. *Structure of arrays (SOA) data layouts*: for problems that require separate subvectors for each solution component. For example, in an incompressible Navier-Stokes simulation, separate subvectors may be used for velocities and pressure, which are combined together into a single NVECTOR_MANYVECTOR for the overall "solution".

The above use cases are neither exhaustive nor mutually exclusive, and the NVECTOR_MANYVECTOR implementation should support arbitrary combinations of these cases.

The NVECTOR_MANYVECTOR implementation is designed to work with any NVECTOR subvectors that implement the minimum "standard" set of operations in §6.2.1. Additionally, NVECTOR_MANYVECTOR 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_MANYVECTOR. However, even at present we anticipate that users will be able to leverage such data partitioning in their problem-defining ODE right-hand side function, DAE or nonlinear solver residual function, preconditioners, or custom *SUNLinearSolver* or `SUNNonlinearSolver` modules.

6.16.1 NVECTOR_MANYVECTOR structure

The NVECTOR_MANYVECTOR implementation defines the *content* field of `N_Vector` to be a structure containing the number of subvectors comprising the ManyVector, the global length of the ManyVector (including all subvectors), 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_ManyVector {
  sunindextype  num_subvectors;  /* number of vectors attached      */
  sunindextype  global_length;   /* overall manyvector length       */
  N_Vector*     subvec_array;    /* pointer to N_Vector array       */
  booleantype   own_data;        /* flag indicating data ownership  */
};
```

The header file to include when using this module is `nvector_manyvector.h`. The installed module library to link against is `libsundials_nvecmanyvector.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

6.16.2 NVECTOR_MANYVECTOR functions

The NVECTOR_MANYVECTOR module implements all vector operations listed in §6.2 except for *N_VGetArrayPointer()*, *N_VSetArrayPointer()*, *N_VScaleAddMultiVectorArray()*, and *N_VLinearCombinationVectorArray()*. As such, this vector cannot be used with the SUNDIALS direct solvers and preconditioners. Instead, the NVECTOR_MANYVECTOR 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 §6.2 by appending the suffix `_ManyVector` (e.g. `N_VDestroy_ManyVector`). The module NVECTOR_MANYVECTOR provides the following additional user-callable routines:

N_Vector **N_VNew_ManyVector**(*sunindextype* num_subvectors, *N_Vector* *vec_array, *SUNContext* sunctx)
: 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 §4.4.2.5, and the functions are documented in §6.1.1.

N_Vector **N_VGetSubvector_ManyVector**(*N_Vector* v, *sunindextype* vec_num)
: This function returns the *vec_num* subvector from the NVECTOR array.

realtype ***N_VGetSubvectorArrayPointer_ManyVector**(*N_Vector* v, *sunindextype* vec_num)
: 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.

int **N_VSetSubvectorArrayPointer_ManyVector**(*realtype* *v_data, *N_Vector* v, *sunindextype* vec_num)
: 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 `-1` is returned; otherwise it returns `0`.

sunindextype **N_VGetNumSubvectors_ManyVector**(*N_Vector* v)
This function returns the overall number of subvectors in the ManyVector object.

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_VNew_ManyVector()*, 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_ManyVector()* 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_VNew_ManyVector()*.

int **N_VEnableFusedOps_ManyVector**(*N_Vector* v, *booleantype* tf)
This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the manyvector vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearCombination_ManyVector**(*N_Vector* v, *booleantype* tf)
This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the manyvector vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleAddMulti_ManyVector**(*N_Vector* v, *booleantype* tf)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the manyvector vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

int **N_VEnableDotProdMulti_ManyVector**(*N_Vector* v, *booleantype* tf)
This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the manyvector vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearSumVectorArray_ManyVector**(*N_Vector* v, *booleantype* tf)
This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the manyvector vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleVectorArray_ManyVector**(*N_Vector* v, *booleantype* tf)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the manyvector vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

int **N_VEnableConstVectorArray_ManyVector**(*N_Vector* v, *booleantype* tf)
This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the manyvector vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

int **N_VEnableWrmsNormVectorArray_ManyVector**(*N_Vector* v, *booleantype* tf)
This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the manyvector vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

int **N_VEnableWrmsNormMaskVectorArray_ManyVector**(*N_Vector* v, *booleantype* tf)
This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the manyvector vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

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.

6.17 The NVECTOR_MPIMANYVECTOR Module

The NVECTOR_MPIMANYVECTOR module is designed to facilitate problems with an inherent data partitioning for the solution vector, and when using distributed-memory parallel architectures. As such, this implementation supports all use cases allowed by the MPI-unaware NVECTOR_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. Three potential use cases for this module include:

A. *Heterogenous computational architectures (single-node or multi-node)*: for data partitioning between different computing resources on a node, architecture-specific subvectors may be created for each partition. For example, a user could create one MPI-parallel component based on *NVECTOR_PARALLEL*, another GPU-accelerated component based on *NVECTOR_CUDA*.

B. *Process-based multiphysics decompositions (multi-node)*: for computations that combine separate MPI-based simulations together, each subvector may reside on a different MPI communicator, and the MPIManyVector combines these via an MPI *intercommunicator* that connects these distinct simulations together.

C. *Structure of arrays (SOA) data layouts (single-node or multi-node)*: for problems that require separate subvectors for each solution component. For example, in an incompressible Navier-Stokes simulation, separate subvectors may be used for velocities and pressure, which are combined together into a single MPIManyVector for the overall "solution".

The above use cases are neither exhaustive nor mutually exclusive, and the NVECTOR_MANYVECTOR implementation should support arbitrary combinations of these cases.

The NVECTOR_MPIMANYVECTOR implementation is designed to work with any NVECTOR subvectors that implement the minimum "standard" set of operations in §6.2.1, however significant performance benefits may be obtained when subvectors additionally implement the optional local reduction operations listed in §6.2.4.

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 function, DAE or nonlinear solver residual function, preconditioners, or custom *SUNLinearSolver* or `SUNNonlinearSolver` modules.

6.17.1 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 ranks), 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       */
  booleantype   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.

6.17.2 NVECTOR_MPIMANYVECTOR functions

The NVECTOR_MPIMANYVECTOR module implements all vector operations listed in §6.2, except for *N_VGetArrayPointer()*, *N_VSetArrayPointer()*, *N_VScaleAddMultiVectorArray()*, and *N_VLinearCombinationVectorArray()*. As such, this vector cannot be used 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 §6.2 by appending the suffix `_MPIManyVector` (e.g. `N_VDestroy_MPIManyVector`). The module NVECTOR_MPIMANYVECTOR provides the following additional user-callable routines:

N_Vector **N_VNew_MPIManyVector**(*sunindextype* num_subvectors, *N_Vector* *vec_array, *SUNContext* sunctx)

This function creates a 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 §4.4.2.5, and the functions are documented in §6.1.1.

N_Vector **N_VMake_MPIManyVector**(MPI_Comm comm, *sunindextype* num_subvectors, *N_Vector* *vec_array, *SUNContext* sunctx)

This function creates a 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`).

N_Vector **N_VGetSubvector_MPIManyVector**(*N_Vector* v, *sunindextype* vec_num)

This function returns the *vec_num* subvector from the NVECTOR array.

realtype ***N_VGetSubvectorArrayPointer_MPIManyVector**(*N_Vector* v, *sunindextype* vec_num)

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.

int **N_VSetSubvectorArrayPointer_MPIManyVector**(*realtype* *v_data, *N_Vector* v, *sunindextype* vec_num)

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 `-1` is returned; otherwise it returns `0`.

sunindextype **N_VGetNumSubvectors_MPIManyVector**(*N_Vector* v)

This function returns the overall number of subvectors in the MPIManyVector object.

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()*.

int **N_VEnableFusedOps_MPIManyVector**(*N_Vector* v, *booleantype* tf)

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the MPIManyVector vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableLinearCombination_MPIManyVector**(*N_Vector* v, *booleantype* tf)

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the MPIManyVector vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableScaleAddMulti_MPIManyVector**(*N_Vector* v, *booleantype* tf)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the MPIManyVector vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

int **N_VEnableDotProdMulti_MPIManyVector**(*N_Vector* v, *booleantype* tf)
This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the MPIManyVector vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearSumVectorArray_MPIManyVector**(*N_Vector* v, *booleantype* tf)
This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the MPIManyVector vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleVectorArray_MPIManyVector**(*N_Vector* v, *booleantype* tf)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the MPIManyVector vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

int **N_VEnableConstVectorArray_MPIManyVector**(*N_Vector* v, *booleantype* tf)
This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the MPIManyVector vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

int **N_VEnableWrmsNormVectorArray_MPIManyVector**(*N_Vector* v, *booleantype* tf)
This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the MPIManyVector vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

int **N_VEnableWrmsNormMaskVectorArray_MPIManyVector**(*N_Vector* v, *booleantype* tf)
This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the MPIManyVector vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

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.

6.18 The NVECTOR_MPIPLUSX Module

The NVECTOR_MPIPLUSX module 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 "standard" set of operations in §6.2.1. 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.

6.18.1 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 §6.17.1.

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.

6.18.2 NVECTOR_MPIPLUSX functions

The NVECTOR_MPIPLUSX module adopts all vector operations listed in §6.2, from the NVECTOR_MPIMANYVECTOR (see §6.17) 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, it's compatibility 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_Vector **N_VMake_MPIPlusX**(MPI_Comm comm, *N_Vector* *local_vector, *SUNContext* sunctx)
: This function creates a MPIPlusX vector from an exisiting 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 NVECTOR 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`).

N_Vector **N_VGetLocal_MPIPlusX**(*N_Vector* v)
: This function returns the local vector underneath the MPIPlusX NVECTOR.

realtype ***N_VGetArrayPointer_MPIPlusX**(*N_Vector* v)
: This function returns the data array pointer for the local vector.

  If the local vector does not support the *N_VGetArrayPointer()* operation, then `NULL` is returned.

void **N_VSetArrayPointer_MPIPlusX**(*realtype* *v_data, *N_Vector* v)
: This function sets the data array pointer for the local vector if the local vector implements the *N_VSetArrayPointer()* operation.

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 a 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 subvector representations.

6.19 NVECTOR Examples

There are NVECTOR examples that may be installed for eac himplementation. 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_VInv`: Compute z[i] = 1 / x[i]

** `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_VConstVectorArray`: Case 1a: z = c
- `Test_N_VConstVectorArray`: 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.
- `Test_N_VMBufSize`: Tests for accuracy in the reported buffer size.
- `Test_N_VMBufPack`: Tests for accuracy in the buffer packing routine.
- `Test_N_VMBufUnpack`: Tests for accuracy in the buffer unpacking routine.

Chapter 7

Matrix Data Structures

The SUNDIALS library comes packaged with a variety of `SUNMatrix` implementations, designed for simulations requiring direct linear solvers for problems in serial or shared-memory parallel environments. SUNDIALS additionally provides a simple interface for generic matrices (akin to a C++ *abstract base class*). All of the major SUNDIALS packages (CVODE(s), IDA(s), KINSOL, ARKODE), are constructed to only depend on these generic matrix operations, making them immediately extensible to new user-defined matrix objects. For each of the SUNDIALS-provided matrix types, SUNDIALS also provides at least two `SUNLinearSolver` implementations that factor these matrix objects and use them in the solution of linear systems.

7.1 Description of the SUNMATRIX Modules

For problems that involve direct methods for solving linear systems, the SUNDIALS packages 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 `N_Vector` and/or linear solver modules, and require matrices that are compatible with those implementations. The generic `SUNMatrix` operations are described below, and descriptions of the SUNMATRIX implementations provided with SUNDIALS follow.

The generic `SUNMatrix` type has been modeled after the object-oriented style of the generic *`N_Vector`* type. Specifically, a generic `SUNMatrix` is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the matrix, and an *ops* field pointing to a structure with generic matrix operations. The type `SUNMatrix` is defined as:

typedef struct _generic_SUNMatrix ***SUNMatrix**

and the generic structure is defined as

```
struct _generic_SUNMatrix {
    void *content;
    struct _generic_SUNMatrix_Ops *ops;
};
```

Here, the `_generic_SUNMatrix_Ops` structure is essentially a list of function pointers to the various actual matrix operations, and is defined as

```
struct _generic_SUNMatrix_Ops {
  SUNMatrix_ID (*getid)(SUNMatrix);
  SUNMatrix    (*clone)(SUNMatrix);
```

(continues on next page)

(continued from previous page)

```
  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 a `SUNMatrix`. 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));
}
```

§7.2 contains a complete list of all matrix operations defined by the generic SUNMATRIX module. A particular implementation of the SUNMATRIX module must:

- Specify the *content* field of the `SUNMatrix` object.
- Define and implement a minimal subset of the matrix operations. See the documentation for each SUNDIALS package and/or linear 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 as needed for that particular implementation to be used to access different parts in the content field of the newly defined `SUNMatrix`.

To aid in the creation of custom SUNMATRIX modules the generic SUNMATRIX module provides three utility functions *SUNMatNewEmpty()*, *SUNMatCopyOps()*, and *SUNMatFreeEmpty()*. 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.

SUNMatrix **SUNMatNewEmpty**()

This function allocates a new generic `SUNMatrix` object and initializes its content pointer and the function pointers in the operations structure to `NULL`.

Return value: If successful, this function returns a `SUNMatrix` object. If an error occurs when allocating the object, then this routine will return `NULL`.

int **SUNMatCopyOps**(*SUNMatrix* A, *SUNMatrix* B)

This function copies the function pointers in the `ops` structure of `A` into the `ops` structure of `B`.

Arguments:

- *A* – the matrix to copy operations from.
- *B* – the matrix to copy operations to.

Return value: If successful, this function returns `0`. If either of the inputs are `NULL` or the `ops` structure of either input is `NULL`, then is function returns a non-zero value.

void **SUNMatFreeEmpty**(*SUNMatrix* A)

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* – the SUNMatrix object to free

Each SUNMATRIX implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 7.1. It is recommended that a user-supplied SUNMATRIX implementation use the `SUNMATRIX_CUSTOM` identifier.

Table 7.1: Identifiers associated with matrix kernels supplied with SUNDIALS

Matrix ID	Matrix type	ID Value
SUNMATRIX_DENSE	Dense $M \times N$ matrix	0
SUNMATRIX_MAGMADENSE	Magma dense $M \times N$ matrix	1
SUNMATRIX_BAND	Band $M \times M$ matrix	2
SUNMATRIX_SPARSE	Sparse (CSR or CSC) $M \times N$ matrix	3
SUNMATRIX_SLUNRLOC	SUNMatrix wrapper for SuperLU_DIST SuperMatrix	4
SUNMATRIX_CUSPARSE	CUDA sparse CSR matrix	5
SUNMATRIX_CUSTOM	User-provided custom matrix	6

7.2 Description of the SUNMATRIX operations

For each of the `SUNMatrix` operations, we give the name, usage of the function, and a description of its mathematical operations below.

SUNMatrix_ID **SUNMatGetID**(*SUNMatrix* A)

Returns the type identifier for the matrix *A*. It is used to determine the matrix implementation type (e.g. dense, banded, sparse,...) from the abstract `SUNMatrix` interface. This is used to assess compatibility with SUNDIALS-provided linear solver implementations. Returned values are given in Table 7.1

Usage:

```
id = SUNMatGetID(A);
```

SUNMatrix **SUNMatClone**(*SUNMatrix* A)

Creates a new `SUNMatrix` of the same type as an existing matrix *A* and sets the *ops* field. It does not copy the matrix values, but rather allocates storage for the new matrix.

Usage:

```
B = SUNMatClone(A);
```

void **SUNMatDestroy**(*SUNMatrix* A)

Destroys the `SUNMatrix` *A* and frees memory allocated for its internal data.

Usage:

```
SUNMatDestroy(A);
```

int **SUNMatSpace**(*SUNMatrix* A, long int *lrw, long int *liw)
Returns the storage requirements for the matrix *A*. *lrw* contains the number of realtype words and *liw* contains the number of integer words. The return value denotes success/failure of the operation.

This function is advisory only, for use in determining a user's total space requirements; it could be a dummy function in a user-supplied `SUNMatrix` module if that information is not of interest.

Usage:

```
retval = SUNMatSpace(A, &lrw, &liw);
```

int **SUNMatZero**(*SUNMatrix* A)
Zeros all entries of the `SUNMatrix` *A*. The return value is an integer flag denoting success/failure of the operation:

$$A_{i,j} = 0, \quad i = 1, \ldots, m, \ j = 1, \ldots, n.$$

Usage:

```
retval = SUNMatZero(A);
```

int **SUNMatCopy**(*SUNMatrix* A, *SUNMatrix* B)
Performs the operation *B gets A* for all entries of the matrices *A* and *B*. The return value is an integer flag denoting success/failure of the operation:

$$B_{i,j} = A_{i,j}, \quad i = 1, \ldots, m, \ j = 1, \ldots, n.$$

Usage:

```
retval = SUNMatCopy(A,B);
```

int **SUNMatScaleAdd**(*realtype* c, *SUNMatrix* A, *SUNMatrix* B)
Performs the operation *A gets cA + B*. The return value is an integer flag denoting success/failure of the operation:

$$A_{i,j} = cA_{i,j} + B_{i,j}, \quad i = 1, \ldots, m, \ j = 1, \ldots, n.$$

Usage:

```
retval = SUNMatScaleAdd(c, A, B);
```

int **SUNMatScaleAddI**(*realtype* c, *SUNMatrix* A)
Performs the operation *A gets cA + I*. The return value is an integer flag denoting success/failure of the operation:

$$A_{i,j} = cA_{i,j} + \delta_{i,j}, \quad i, j = 1, \ldots, n.$$

Usage:

```
retval = SUNMatScaleAddI(c, A);
```

int **SUNMatMatvecSetup**(*SUNMatrix* A)
Performs any setup necessary to perform a matrix-vector product. The return value is an integer flag denoting success/failure of the operation. It is useful for SUNMatrix implementations which need to prepare the matrix itself, or communication structures before performing the matrix-vector product.

Usage:

```
retval = SUNMatMatvecSetup(A);
```

int **SUNMatMatvec**(*SUNMatrix* A, *N_Vector* x, *N_Vector* y)

Performs the matrix-vector product *y gets Ax*. It should only be called with vectors *x* and *y* that are compatible with the matrix *A* – both in storage type and dimensions. The return value is an integer flag denoting success/failure of the operation:

$$y_i = \sum_{j=1}^{n} A_{i,j} x_j, \quad i = 1, \ldots, m.$$

Usage:

```
retval = SUNMatMatvec(A, x, y);
```

7.2.1 SUNMatrix return codes

The functions provided to SUNMatrix modules within the SUNDIALS-provided SUNMatrix implementations utilize a common set of return codes, listed below. These adhere to a common pattern: 0 indicates success, a negative value indicates a 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 SUNMatrix failure.

- `SUNMAT_SUCCESS` (0) – successful call
- `SUNMAT_ILL_INPUT` (-1) – an illegal input has been provided to the function
- `SUNMAT_MEM_FAIL` (-2) – failed memory access or allocation
- `SUNMAT_OPERATION_FAIL` (-3) – a SUNMatrix operation returned nonzero
- `SUNMAT_MATVEC_SETUP_REQUIRED` (-4) – the *SUNMatMatvecSetup()* routine needs to be called prior to calling *SUNMatMatvec()*

7.3 The SUNMATRIX_DENSE Module

The dense implementation of the `SUNMatrix` module, 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) element of a dense `SUNMatrix` object (with $0 \le i < M$ and $0 \le j < N$) may be accessed via `data[j*M+i]`.
- `ldata` - length of the data array ($= M\,N$).

- `cols` - array of pointers. `cols[j]` points to the first element of the j-th column of the matrix in the array `data`. The (i, j) element of a dense `SUNMatrix` (with $0 \leq i < M$ and $0 \leq j < N$) may be accessed may be accessed via `cols[j][i]`.

The header file to be included when using this module is `sunmatrix/sunmatrix_dense.h`.

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(A)
This macro gives access to the contents of the dense `SUNMatrix` A.

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(A)
Access the number of rows in the dense `SUNMatrix` A.

This may be used either to retrieve or to set the value. 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_ROWS_D(A) = A_rows` sets the number of columns in `A` to equal `A_rows`.

Implementation:

```
#define SM_ROWS_D(A)   ( SM_CONTENT_D(A)->M )
```

SM_COLUMNS_D(A)
Access the number of columns in the dense `SUNMatrix` A.

This may be used either to retrieve or to set the value. For example, the assignment `A_columns = SM_COLUMNS_D(A)` sets `A_columns` to be the number of columns in the matrix `A`. Similarly, the assignment `SM_COLUMNS_D(A) = A_columns` sets the number of columns in `A` to equal `A_columns`

Implementation:

```
#define SM_COLUMNS_D(A)   ( SM_CONTENT_D(A)->N )
```

SM_LDATA_D(A)
Access the total data length in the dense `SUNMatrix` A.

This may be used either to retrieve or to set the value. For example, the assignment `A_ldata = SM_LDATA_D(A)` sets `A_ldata` to be the length of the data array in the matrix `A`. Similarly, the assignment `SM_LDATA_D(A) = A_ldata` sets the parameter for the length of the data array in `A` to equal `A_ldata`.

Implementation:

```
#define SM_LDATA_D(A)   ( SM_CONTENT_D(A)->ldata )
```

SM_DATA_D(A)
This macro gives access to the `data` pointer 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`.

Implementation:

```
#define SM_DATA_D(A)   ( SM_CONTENT_D(A)->data )
```

SM_COLS_D(A)
This macro gives access to the `cols` pointer for the matrix entries.

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_COLS_D(A)   ( SM_CONTENT_D(A)->cols )
```

SM_COLUMN_D(A)
This macros gives access to the individual columns 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`.

Implementation:

```
#define SM_COLUMN_D(A,j)    ( (SM_CONTENT_D(A)->cols)[j] )
```

SM_ELEMENT_D(A)
This macro gives access to the individual entries of the data array of a dense `SUNMatrix`.

The assignments `SM_ELEMENT_D(A,i,j) = a_ij` and `a_ij = SM_ELEMENT_D(A,i,j)` reference the $A_{i,j}$ element of the $M \times N$ dense matrix `A` (with $0 \leq i < M$ and $0 \leq j < N$).

Implementation:

```
#define SM_ELEMENT_D(A,i,j) ( (SM_CONTENT_D(A)->cols)[j][i] )
```

The SUNMATRIX_DENSE module defines dense implementations of all matrix operations listed in §7.2. Their names are obtained from those in that section by appending the suffix `_Dense` (e.g. `SUNMatCopy_Dense`). The module SUNMATRIX_DENSE provides the following additional user-callable routines:

SUNMatrix **SUNDenseMatrix**(*sunindextype* M, *sunindextype* N, *SUNContext* sunctx)
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.

void **SUNDenseMatrix_Print**(*SUNMatrix* A, FILE *outfile)
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.

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

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

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

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

realtype ****SUNDenseMatrix_Cols**(*SUNMatrix* A)
This function returns a pointer to the cols array for the dense `SUNMatrix`.

realtype ***SUNDenseMatrix_Column**(*SUNMatrix* A, *sunindextype* j)
This function returns a pointer to the first entry of the jth column of the dense `SUNMatrix`. The resulting pointer should be indexed over the range `0` to `M-1`.

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 = SUNDenseMatrix_Data(A)`, or equivalently `A_data = SM_DATA_D(A)`, and then access `A_data[i]` within the loop.
  - First obtain the array of column pointers via `A_cols = SUNDenseMatrix_Cols(A)`, or equivalently `A_cols = SM_COLS_D(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 `N_Vector` 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.

7.4 The SUNMATRIX_MAGMADENSE Module

The SUNMATRIX_MAGMADENSE module interfaces to the MAGMA linear algebra library and can target NVIDIA's CUDA programming model or AMD's HIP programming model [32]. 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

$$\mathbf{A} = \begin{bmatrix} \mathbf{A_0} & 0 & \cdots & 0 \\ 0 & \mathbf{A_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A_{n-1}} \end{bmatrix}$$

This matrix implementation is best paired with the *SUNLinearSolver_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.

Warning: The SUNMATRIX_MAGMADENSE module is experimental and subject to change.

7.4.1 SUNMATRIX_MAGMADENSE Functions

The SUNMATRIX_MAGMADENSE module defines GPU-enabled implementations of all matrix operations listed in §7.2.

- `SUNMatGetID_MagmaDense` – returns `SUNMATRIX_MAGMADENSE`
- `SUNMatClone_MagmaDense`
- `SUNMatDestroy_MagmaDense`
- `SUNMatZero_MagmaDense`
- `SUNMatCopy_MagmaDense`
- `SUNMatScaleAdd_MagmaDense`
- `SUNMatScaleAddI_MagmaDense`
- `SUNMatMatvecSetup_MagmaDense`
- `SUNMatMatvec_MagmaDense`
- `SUNMatSpace_MagmaDense`

In addition, the SUNMATRIX_MAGMADENSE module defines the following implementation specific functions:

SUNMatrix **SUNMatrix_MagmaDense**(*sunindextype* M, *sunindextype* N, *SUNMemoryType* memtype, *SUNMemoryHelper* memhelper, void *queue, *SUNContext* sunctx)

This constructor function creates and allocates memory for an $M \times N$ SUNMATRIX_MAGMADENSE `SUNMatrix`.

Arguments:

- *M* – the number of matrix rows.
- *N* – the number of matrix columns.
- *memtype* – the type of memory to use for the matrix data; can be `SUNMEMTYPE_UVM` or `SUNMEMTYPE_DEVICE`.
- *memhelper* – the memory helper used for allocating data.
- *queue* – a `cudaStream_t` when using CUDA or a `hipStream_t` when using HIP.
- *sunctx* – the *SUNContext* object (see §4.1)

Return value: If successful, a `SUNMatrix` object otherwise `NULL`.

SUNMatrix **SUNMatrix_MagmaDenseBlock**(*sunindextype* nblocks, *sunindextype* M_block, *sunindextype* N_block, *SUNMemoryType* memtype, *SUNMemoryHelper* memhelper, void *queue, *SUNContext* sunctx)

This constructor function creates and allocates memory for a block diagonal SUNMATRIX_MAGMADENSE `SUNMatrix` with *nblocks* of size $M \times N$.

Arguments:

- *nblocks* – the number of matrix rows.
- *M_block* – the number of matrix rows in each block.
- *N_block* – the number of matrix columns in each block.
- *memtype* – the type of memory to use for the matrix data; can be `SUNMEMTYPE_UVM` or `SUNMEMTYPE_DEVICE`.
- *memhelper* – the memory helper used for allocating data.

- *queue* – a `cudaStream_t` when using CUDA or a `hipStream_t` when using HIP.
- *sunctx* – the *SUNContext* object (see §4.1)

Return value: If successful, a `SUNMatrix` object otherwise `NULL`.

sunindextype **SUNMatrix_MagmaDense_Rows**(*SUNMatrix* A)
This function returns the number of rows in the `SUNMatrix` object. For block diagonal matrices, the number of rows is computed as $M_{\text{block}} \times$ nblocks.

Arguments:

- *A* – a `SUNMatrix` object.

Return value: If successful, the number of rows in the `SUNMatrix` object otherwise `SUNMATRIX_ILL_INPUT`.

sunindextype **SUNMatrix_MagmaDense_Columns**(*SUNMatrix* A)
This function returns the number of columns in the `SUNMatrix` object. For block diagonal matrices, the number of columns is computed as $N_{\text{block}} \times$ nblocks.

Arguments:

- *A* – a `SUNMatrix` object.

Return value: If successful, the number of columns in the `SUNMatrix` object otherwise `SUNMATRIX_ILL_INPUT`.

sunindextype **SUNMatrix_MagmaDense_BlockRows**(*SUNMatrix* A)
This function returns the number of rows in a block of the `SUNMatrix` object.

Arguments:

- *A* – a `SUNMatrix` object.

Return value: If successful, the number of rows in a block of the `SUNMatrix` object otherwise `SUNMATRIX_ILL_INPUT`.

sunindextype **SUNMatrix_MagmaDense_BlockColumns**(*SUNMatrix* A)
This function returns the number of columns in a block of the `SUNMatrix` object.

Arguments:

- *A* – a `SUNMatrix` object.

Return value: If successful, the number of columns in a block of the `SUNMatrix` object otherwise `SUNMATRIX_ILL_INPUT`.

sunindextype **SUNMatrix_MagmaDense_LData**(*SUNMatrix* A)
This function returns the length of the `SUNMatrix` data array.

Arguments:

- *A* – a `SUNMatrix` object.

Return value: If successful, the length of the `SUNMatrix` data array otherwise `SUNMATRIX_ILL_INPUT`.

sunindextype **SUNMatrix_MagmaDense_NumBlocks**(*SUNMatrix* A)
This function returns the number of blocks in the `SUNMatrix` object.

Arguments:

- *A* – a `SUNMatrix` object.

Return value: If successful, the number of blocks in the `SUNMatrix` object otherwise `SUNMATRIX_ILL_INPUT`.

realtype ***SUNMatrix_MagmaDense_Data**(*SUNMatrix* A)
This function returns the `SUNMatrix` data array.

Arguments:

- *A* – a `SUNMatrix` object.

Return value: If successful, the `SUNMatrix` data array otherwise `NULL`.

realtype ****SUNMatrix_MagmaDense_BlockData**(*SUNMatrix* A)**
This function returns an array of pointers that point to the start of the data array for each block in the `SUNMatrix`.

Arguments:

- *A* – a `SUNMatrix` object.

Return value: If successful, an array of data pointers to each of the `SUNMatrix` blocks otherwise `NULL`.

realtype ***SUNMatrix_MagmaDense_Block**(*SUNMatrix* A, *sunindextype* k)**
This function returns a pointer to the data array for block *k* in the `SUNMatrix`.

Arguments:

- *A* – a `SUNMatrix` object.
- *k* – the block index.

Return value: If successful, a pointer to the data array for the `SUNMatrix` block otherwise `NULL`.

Note: No bounds-checking is performed by this function, *j* should be strictly less than *nblocks*.

---

realtype ***SUNMatrix_MagmaDense_Column**(*SUNMatrix* A, *sunindextype* j)**
This function returns a pointer to the data array for column *j* in the `SUNMatrix`.

Arguments:

- *A* – a `SUNMatrix` object.
- *j* – the column index.

Return value: If successful, a pointer to the data array for the `SUNMatrix` column otherwise `NULL`.

Note: No bounds-checking is performed by this function, *j* should be strictly less than $nblocks * N_{\text{block}}$.

---

realtype ***SUNMatrix_MagmaDense_BlockColumn**(*SUNMatrix* A, *sunindextype* k, *sunindextype* j)**
This function returns a pointer to the data array for column *j* of block *k* in the `SUNMatrix`.

Arguments:

- *A* – a `SUNMatrix` object.
- *k* – the block index.
- *j* – the column index.

Return value: If successful, a pointer to the data array for the `SUNMatrix` column otherwise `NULL`.

Note: No bounds-checking is performed by this function, *k* should be strictly less than *nblocks* and *j* should be strictly less than N_{block}.

---

int **SUNMatrix_MagmaDense_CopyToDevice**(*SUNMatrix* A, *realtype* *h_data)**
This function copies the matrix data to the GPU device from the provided host array.

Arguments:

- *A* – a `SUNMatrix` object
- *h_data* – a host array pointer to copy data from.

Return value:

- `SUNMAT_SUCCESS` – if the copy is successful.
- `SUNMAT_ILL_INPUT` – if either the `SUNMatrix` is not a `SUNMATRIX_MAGMADENSE` matrix.
- `SUNMAT_MEM_FAIL` – if the copy fails.

int **`SUNMatrix_MagmaDense_CopyFromDevice`**(*SUNMatrix* A, *realtype* *h_data)

This function copies the matrix data from the GPU device to the provided host array.

Arguments:

- *A* – a `SUNMatrix` object
- *h_data* – a host array pointer to copy data to.

Return value:

- `SUNMAT_SUCCESS` – if the copy is successful.
- `SUNMAT_ILL_INPUT` – if either the `SUNMatrix` is not a `SUNMATRIX_MAGMADENSE` matrix.
- `SUNMAT_MEM_FAIL` – if the copy fails.

7.4.2 SUNMATRIX_MAGMADENSE Usage Notes

Warning: 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.

7.5 The SUNMATRIX_ONEMKLDENSE Module

The SUNMATRIX_ONEMKLDENSE module 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* linear solver.

The header file to include when using this class is `sunmatrix/sunmatrix_onemkldense.h`. The installed library to link to is `libsundials_sunmatrixonemkldense.lib` where `lib` is typically `.so` for shared libraries and `.a` for static libraries.

Warning: The SUNMATRIX_ONEMKLDENSE class is experimental and subject to change.

7.5.1 SUNMATRIX_ONEMKLDENSE Functions

The SUNMATRIX_ONEMKLDENSE class defines implementations of the following matrix operations listed in §7.2.

- SUNMatGetID_OneMklDense – returns SUNMATRIX_ONEMKLDENSE
- SUNMatClone_OneMklDense
- SUNMatDestroy_OneMklDense
- SUNMatZero_OneMklDense
- SUNMatCopy_OneMklDense
- SUNMatScaleAdd_OneMklDense
- SUNMatScaleAddI_OneMklDense
- SUNMatMatvec_OneMklDense
- SUNMatSpace_OneMklDense

In addition, the SUNMATRIX_ONEMKLDENSE class defines the following implementation specific functions.

7.5.1.1 Constructors

SUNMatrix **SUNMatrix_OneMklDense**(sunindextype M, sunindextype N, SUNMemoryType memtype, SUNMemoryHelper memhelper, sycl::queue *queue, SUNContext sunctx)

This constructor function creates and allocates memory for an $M \times N$ SUNMATRIX_ONEMKLDENSE SUNMatrix.

Arguments:

- *M* – the number of matrix rows.
- *N* – the number of matrix columns.
- *memtype* – the type of memory to use for the matrix data; can be SUNMEMTYPE_UVM or SUNMEMTYPE_DEVICE.
- *memhelper* – the memory helper used for allocating data.
- *queue* – the SYCL queue to which operations will be submitted.
- *sunctx* – the *SUNContext* object (see §4.1)

Return value: If successful, a SUNMatrix object otherwise NULL.

SUNMatrix **SUNMatrix_OneMklDenseBlock**(sunindextype nblocks, sunindextype M_block, sunindextype N_block, SUNMemoryType memtype, SUNMemoryHelper memhelper, sycl::queue *queue, SUNContext sunctx)

This constructor function creates and allocates memory for a block diagonal SUNMATRIX_ONEMKLDENSE SUNMatrix with *nblocks* of size $M_{block} \times N_{block}$.

Arguments:

- *nblocks* – the number of matrix rows.
- *M_block* – the number of matrix rows in each block.
- *N_block* – the number of matrix columns in each block.
- *memtype* – the type of memory to use for the matrix data; can be SUNMEMTYPE_UVM or SUNMEMTYPE_DEVICE.
- *memhelper* – the memory helper used for allocating data.

- *queue* – the SYCL queue to which operations will be submitted.
- *sunctx* – the `SUNContext` object (see §4.1)

Return value: If successful, a `SUNMatrix` object otherwise `NULL`.

7.5.1.2 Access Matrix Dimensions

sunindextype **SUNMatrix_OneMklDense_Rows**(*SUNMatrix* A)
This function returns the number of rows in the `SUNMatrix` object. For block diagonal matrices, the number of rows is computed as $M_{\text{block}} \times$ nblocks.

Arguments:

- *A* – a `SUNMatrix` object.

Return value: If successful, the number of rows in the `SUNMatrix` object otherwise `SUNMATRIX_ILL_INPUT`.

sunindextype **SUNMatrix_OneMklDense_Columns**(*SUNMatrix* A)
This function returns the number of columns in the `SUNMatrix` object. For block diagonal matrices, the number of columns is computed as $N_{\text{block}} \times$ nblocks.

Arguments:

- *A* – a `SUNMatrix` object.

Return value: If successful, the number of columns in the `SUNMatrix` object otherwise `SUNMATRIX_ILL_INPUT`.

7.5.1.3 Access Matrix Block Dimensions

sunindextype **SUNMatrix_OneMklDense_NumBlocks**(*SUNMatrix* A)
This function returns the number of blocks in the `SUNMatrix` object.

Arguments:

- *A* – a `SUNMatrix` object.

Return value: If successful, the number of blocks in the `SUNMatrix` object otherwise `SUNMATRIX_ILL_INPUT`.

sunindextype **SUNMatrix_OneMklDense_BlockRows**(*SUNMatrix* A)
This function returns the number of rows in a block of the `SUNMatrix` object.

Arguments:

- *A* – a `SUNMatrix` object.

Return value: If successful, the number of rows in a block of the `SUNMatrix` object otherwise `SUNMATRIX_ILL_INPUT`.

sunindextype **SUNMatrix_OneMklDense_BlockColumns**(*SUNMatrix* A)
This function returns the number of columns in a block of the `SUNMatrix` object.

Arguments:

- *A* – a `SUNMatrix` object.

Return value: If successful, the number of columns in a block of the `SUNMatrix` object otherwise `SUNMATRIX_ILL_INPUT`.

7.5.1.4 Access Matrix Data

sunindextype **SUNMatrix_OneMklDense_LData**(*SUNMatrix* A)
This function returns the length of the SUNMatrix data array.

Arguments:

- *A* – a SUNMatrix object.

Return value: If successful, the length of the SUNMatrix data array otherwise SUNMATRIX_ILL_INPUT.

realtype ***SUNMatrix_OneMklDense_Data**(*SUNMatrix* A)
This function returns the SUNMatrix data array.

Arguments:

- *A* – a SUNMatrix object.

Return value: If successful, the SUNMatrix data array otherwise NULL.

realtype ***SUNMatrix_OneMklDense_Column**(*SUNMatrix* A, *sunindextype* j)
This function returns a pointer to the data array for column *j* in the SUNMatrix.

Arguments:

- *A* – a SUNMatrix object.
- *j* – the column index.

Return value: If successful, a pointer to the data array for the SUNMatrix column otherwise NULL.

Note: No bounds-checking is performed by this function, *j* should be strictly less than $nblocks * N_{\text{block}}$.

7.5.1.5 Access Matrix Block Data

sunindextype **SUNMatrix_OneMklDense_BlockLData**(*SUNMatrix* A)
This function returns the length of the SUNMatrix data array for each block of the SUNMatrix object.

Arguments:

- *A* – a SUNMatrix object.

Return value: If successful, the length of the SUNMatrix data array for each block otherwise SUNMATRIX_ILL_INPUT.

realtype ****SUNMatrix_OneMklDense_BlockData**(*SUNMatrix* A)
This function returns an array of pointers that point to the start of the data array for each block in the SUNMatrix.

Arguments:

- *A* – a SUNMatrix object.

Return value: If successful, an array of data pointers to each of the SUNMatrix blocks otherwise NULL.

realtype ***SUNMatrix_OneMklDense_Block**(*SUNMatrix* A, *sunindextype* k)
This function returns a pointer to the data array for block *k* in the SUNMatrix.

Arguments:

- *A* – a SUNMatrix object.
- *k* – the block index.

Return value: If successful, a pointer to the data array for the SUNMatrix block otherwise NULL.

Note: No bounds-checking is performed by this function, *j* should be strictly less than *nblocks*.

realtype ***SUNMatrix_OneMklDense_BlockColumn**(*SUNMatrix* A, *sunindextype* k, *sunindextype* j)
This function returns a pointer to the data array for column *j* of block *k* in the `SUNMatrix`.

Arguments:

- *A* – a `SUNMatrix` object.
- *k* – the block index.
- *j* – the column index.

Return value: If successful, a pointer to the data array for the `SUNMatrix` column otherwise `NULL`.

Note: No bounds-checking is performed by this function, *k* should be strictly less than *nblocks* and *j* should be strictly less than N_{block}.

7.5.1.6 Copy Data

int **SUNMatrix_OneMklDense_CopyToDevice**(*SUNMatrix* A, *realtype* *h_data)
This function copies the matrix data to the GPU device from the provided host array.

Arguments:

- *A* – a `SUNMatrix` object
- *h_data* – a host array pointer to copy data from.

Return value:

- `SUNMAT_SUCCESS` – if the copy is successful.
- `SUNMAT_ILL_INPUT` – if either the `SUNMatrix` is not a `SUNMATRIX_ONEMKLDENSE` matrix.
- `SUNMAT_MEM_FAIL` – if the copy fails.

int **SUNMatrix_OneMklDense_CopyFromDevice**(*SUNMatrix* A, *realtype* *h_data)
This function copies the matrix data from the GPU device to the provided host array.

Arguments:

- *A* – a `SUNMatrix` object
- *h_data* – a host array pointer to copy data to.

Return value:

- `SUNMAT_SUCCESS` – if the copy is successful.
- `SUNMAT_ILL_INPUT` – if either the `SUNMatrix` is not a `SUNMATRIX_ONEMKLDENSE` matrix.
- `SUNMAT_MEM_FAIL` – if the copy fails.

7.5.2 SUNMATRIX_ONEMKLDENSE Usage Notes

Warning: 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 queue 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 *SUNMatMatvec()* operation. If different streams are utilized, synchronization issues may occur.

7.6 The SUNMATRIX_BAND Module

The banded implementation of the `SUNMatrix` module, 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 smu;
  sunindextype ldim;
  realtype *data;
  sunindextype ldata;
  realtype **cols;
};
```

A diagram of the underlying data representation in a banded matrix is shown in Fig. 7.1. A more complete description of the parts of this *content* field is given below:

- `M` - number of rows
- `N` - number of columns (`N` = `M`)
- `mu` - upper half-bandwidth, $0 \leq$ mu $< N$
- `ml` - lower half-bandwidth, $0 \leq$ ml $< N$
- `smu` - storage upper bandwidth, mu $\leq$ smu $< N$. The LU decomposition routines in the associated *SUNLINSOL_BAND* and *SUNLINSOL_LAPACKBAND* modules write the LU factors into the existing storage for the band matrix. The upper triangular factor U, however, may have an upper bandwidth as big as `min(N-1, mu+ml)` because of partial pivoting. The `smu` field holds the upper half-bandwidth allocated for the band matrix.
- `ldim` - leading dimension (ldim $\geq smu + ml + 1$)
- `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 banded matrix.
- `ldata` - length of the data array (= ldim N)
- `cols` - array of pointers. `cols[j]` is a pointer to the uppermost element within the band in the j-th column. This pointer may be treated as an array indexed from `smu-mu` (to access the uppermost element within the band in the j-th column) to `smu+ml` (to access the lowest element within the band in the j-th column). Indices from 0 to `smu-`

mu-1 give access to extra storage elements required by the LU decomposition function. Finally, cols[j][i-j+smu] is the (i, j)-th element with $j - \text{mu} \leq i \leq j + \text{ml}$.

The header file to be included when using this module is sunmatrix/sunmatrix_band.h.

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(A)
This macro gives access to the contents of the banded SUNMatrix *A*.

The assignment A_cont = SM_CONTENT_B(A) sets A_cont to be a pointer to the banded SUNMatrix content structure.

Implementation:

```
#define SM_CONTENT_B(A)   ( (SUNMatrixContent_Band)(A->content) )
```

SM_ROWS_B(A)
Access the number of rows in the banded SUNMatrix *A*.

This may be used either to retrieve or to set the value. 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_ROWS_B(A) = A_rows sets the number of columns in A to equal A_rows.

Implementation:

```
#define SM_ROWS_B(A)   ( SM_CONTENT_B(A)->M )
```

SM_COLUMNS_B(A)
Access the number of columns in the banded SUNMatrix *A*. As with SM_ROWS_B, this may be used either to retrieve or to set the value.

Implementation:

```
#define SM_COLUMNS_B(A)   ( SM_CONTENT_B(A)->N )
```

SM_UBAND_B(A)
Access the mu parameter in the banded SUNMatrix *A*. As with SM_ROWS_B, this may be used either to retrieve or to set the value.

Implementation:

```
#define SM_UBAND_B(A)   ( SM_CONTENT_B(A)->mu )
```

SM_LBAND_B(A)
Access the ml parameter in the banded SUNMatrix *A*. As with SM_ROWS_B, this may be used either to retrieve or to set the value.

Implementation:

```
#define SM_LBAND_B(A)   ( SM_CONTENT_B(A)->ml )
```

SM_SUBAND_B(A)
Access the smu parameter in the banded SUNMatrix *A*. As with SM_ROWS_B, this may be used either to retrieve or to set the value.

Implementation:

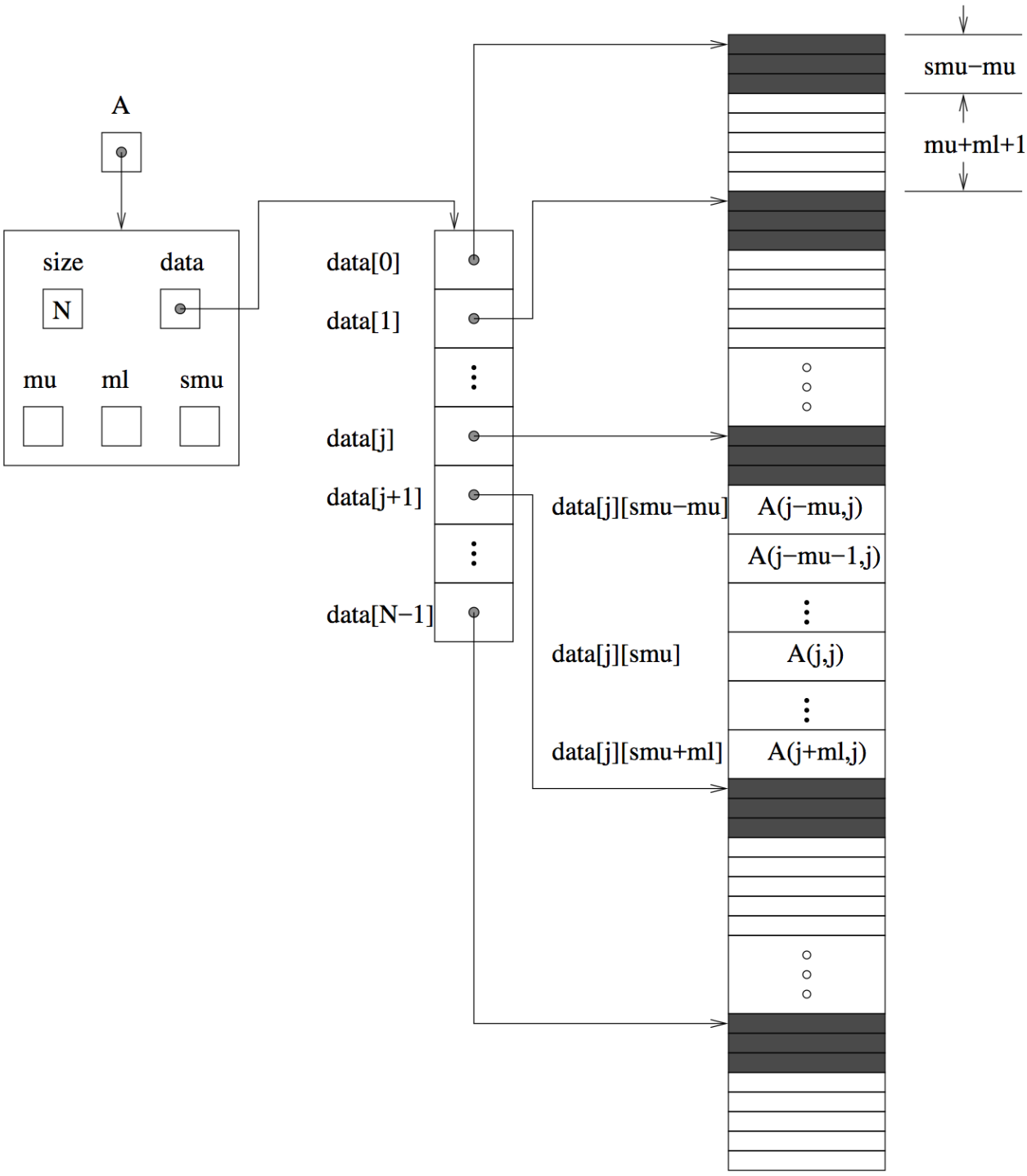

Fig. 7.1: Diagram of the storage for the SUNMATRIX_BAND module. Here `A` is an $N \times N$ band matrix with upper and lower half-bandwidths `mu` and `ml`, respectively. The rows and columns of `A` are numbered from 0 to `N-1` and the (i, j)-th element of `A` is denoted `A(i,j)`. The greyed out areas of the underlying component storage are used by the associated SUNLINSOL_BAND or SUNLINSOL_LAPACKBAND linear solver.

```
#define SM_SUBAND_B(A)   ( SM_CONTENT_B(A)->smu )
```

SM_LDIM_B(A)
Access the ldim parameter in the banded SUNMatrix A. As with SM_ROWS_B, this may be used either to retrieve or to set the value.

Implementation:

```
#define SM_LDIM_B(A)   ( SM_CONTENT_B(A)->ldim )
```

SM_LDATA_B(A)
Access the ldata parameter in the banded SUNMatrix A. As with SM_ROWS_B, this may be used either to retrieve or to set the value.

Implementation:

```
#define SM_LDATA_B(A)   ( SM_CONTENT_B(A)->ldata )
```

SM_DATA_B(A)
This macro gives access to the data pointer 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.

Implementation:

```
#define SM_DATA_B(A)   ( SM_CONTENT_B(A)->data )
```

SM_COLS_B(A)
This macro gives access to the cols pointer for the matrix entries.

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_COLS_B(A)   ( SM_CONTENT_B(A)->cols )
```

SM_COLUMN_B(A)
This macros gives access to the individual columns of the data array of a banded SUNMatrix.

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, $0 \le j \le N-1$. The type of the expression SM_COLUMN_B(A,j) is realtype *. The pointer returned by the call SM_COLUMN_B(A,j) can be treated as an array which is indexed from -mu to ml.

Implementation:

```
#define SM_COLUMN_B(A,j)   ( ((SM_CONTENT_B(A)->cols)[j])+SM_SUBAND_B(A) )
```

SM_ELEMENT_B(A)
This macro gives access to the individual 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 $0 \le i,j \le N-1$. The location (i,j) should further satisfy $j - \text{mu} \le i \le j + \text{ml}$.

Implementation:

```
#define SM_ELEMENT_B(A,i,j)   ( (SM_CONTENT_B(A)->cols)[j][(i)-(j)+SM_SUBAND_B(A)] )
```

SM_COLUMN_ELEMENT_B(A)
This macro gives access to the individual entries of the data array of a banded SUNMatrix.

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 $j - \text{mu} \leq i \leq j + \text{ml}$.

Implementation:

```
#define SM_COLUMN_ELEMENT_B(col_j,i,j)   (col_j[(i)-(j)])
```

The SUNMATRIX_BAND module defines banded implementations of all matrix operations listed in §7.2. Their names are obtained from those in that section by appending the suffix _Band (e.g. SUNMatCopy_Band). The module SUNMATRIX_BAND provides the following additional user-callable routines:

SUNMatrix **SUNBandMatrix**(*sunindextype* N, *sunindextype* mu, *sunindextype* ml, *SUNContext* sunctx)
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.

SUNMatrix **SUNBandMatrixStorage**(*sunindextype* N, *sunindextype* mu, *sunindextype* ml, *sunindextype* smu, *SUNContext* sunctx)
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.

---

void **SUNBandMatrix_Print**(*SUNMatrix* A, FILE *outfile)
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.

sunindextype **SUNBandMatrix_Rows**(*SUNMatrix* A)
This function returns the number of rows in the banded SUNMatrix.

sunindextype **SUNBandMatrix_Columns**(*SUNMatrix* A)
This function returns the number of columns in the banded SUNMatrix.

sunindextype **SUNBandMatrix_LowerBandwidth**(*SUNMatrix* A)
This function returns the lower half-bandwidth for the banded SUNMatrix.

sunindextype **SUNBandMatrix_UpperBandwidth**(*SUNMatrix* A)
This function returns the upper half-bandwidth of the banded SUNMatrix.

sunindextype **SUNBandMatrix_StoredUpperBandwidth**(*SUNMatrix* A)
This function returns the stored upper half-bandwidth of the banded SUNMatrix.

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

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

realtype ****SUNBandMatrix_Cols**(*SUNMatrix* A)
This function returns a pointer to the cols array for the band SUNMatrix.

realtype ***SUNBandMatrix_Column**(*SUNMatrix* A, *sunindextype* j)
This function returns a pointer to the diagonal entry of the j-th column of the banded SUNMatrix. The resulting pointer should be indexed over the range -mu to ml.

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 = SUNBandMatrix_Data(A), or equivalently A_data = SM_DATA_B(A), and then access A_data[i] within the loop.
  - First obtain the array of column pointers via A_cols = SUNBandMatrix_Cols(A), or equivalently A_cols = SM_COLS_B(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 N_Vector 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.

7.7 The SUNMATRIX_CUSPARSE Module

The SUNMATRIX_CUSPARSE module is an interface to the NVIDIA cuSPARSE matrix for use on NVIDIA GPUs [40]. All data stored by this matrix implementation resides on the GPU at all times.

The header file to be included 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.

7.7.1 SUNMATRIX_CUSPARSE Description

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

$$\mathbf{A} = \begin{bmatrix} \mathbf{A_0} & 0 & \cdots & 0 \\ 0 & \mathbf{A_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A_{n-1}} \end{bmatrix},$$

where all the block matrices $\mathbf{A_j}$ 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 the number of blocks is

large. This format is well-suited for, and intended to be used with, the SUNLinearSolver_cuSolverSp_batchQR linear solver (see §8.17).

The SUNMATRIX_CUSPARSE module is experimental and subject to change.

7.7.2 SUNMATRIX_CUSPARSE Functions

The SUNMATRIX_CUSPARSE module defines GPU-enabled sparse implementations of all matrix operations listed in §7.2 except for the *SUNMatSpace()* and *SUNMatMatvecSetup()* operations:

- SUNMatGetID_cuSparse – returns SUNMATRIX_CUSPARSE
- SUNMatClone_cuSparse
- SUNMatDestroy_cuSparse
- SUNMatZero_cuSparse
- SUNMatCopy_cuSparse
- SUNMatScaleAdd_cuSparse – performs $A = cA + B$, where A and B must have the same sparsity pattern
- SUNMatScaleAddI_cuSparse – performs $A = cA + I$, where the diagonal of A must be present
- SUNMatMatvec_cuSparse

In addition, the SUNMATRIX_CUSPARSE module defines the following implementation specific functions:

SUNMatrix **SUNMatrix_cuSparse_NewCSR**(int M, int N, int NNZ, cusparseHandle_t cusp, *SUNContext* sunctx)
This constructor function creates and allocates memory for a SUNMATRIX_CUSPARSE SUNMatrix that uses the CSR storage format. Its arguments are the number of rows and columns of the matrix, M and N, the number of nonzeros to be stored in the matrix, NNZ, and a valid cusparseHandle_t.

SUNMatrix **SUNMatrix_cuSparse_NewBlockCSR**(int nblocks, int blockrows, int blockcols, int blocknnz, cusparseHandle_t cusp, *SUNContext* sunctx)
This constructor function creates and allocates memory for a SUNMATRIX_CUSPARSE SUNMatrix object 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. The function arguments are the number of blocks, nblocks, the number of rows, blockrows, the number of columns, blockcols, the number of nonzeros in each each block, blocknnz, and a valid cusparseHandle_t.

> **Warning:** The SUNMAT_CUSPARSE_BCSR format currently only supports square matrices, i.e., blockrows == blockcols.

SUNMatrix **SUNMatrix_cuSparse_MakeCSR**(cusparseMatDescr_t mat_descr, int M, int N, int NNZ, int *rowptrs, int *colind, *realtype* *data, cusparseHandle_t cusp, *SUNContext* sunctx)
This constructor function creates a SUNMATRIX_CUSPARSE SUNMatrix object from user provided pointers. Its arguments are a cusparseMatDescr_t that must have index base CUSPARSE_INDEX_BASE_ZERO, the number of rows and columns of the matrix, M and N, the number of nonzeros to be stored in the matrix, NNZ, and a valid cusparseHandle_t.

int **SUNMatrix_cuSparse_Rows**(*SUNMatrix* A)
This function returns the number of rows in the sparse SUNMatrix.

int **SUNMatrix_cuSparse_Columns**(*SUNMatrix* A)
This function returns the number of columns in the sparse SUNMatrix.

int **SUNMatrix_cuSparse_NNZ**(*SUNMatrix* A)
This function returns the number of entries allocated for nonzero storage for the sparse SUNMatrix.

int **SUNMatrix_cuSparse_SparseType**(*SUNMatrix* A)
This function returns the storage type (SUNMAT_CUSPARSE_CSR or SUNMAT_CUSPARSE_BCSR) for the sparse SUNMatrix.

realtype ***SUNMatrix_cuSparse_Data**(*SUNMatrix* A)
This function returns a pointer to the data array for the sparse SUNMatrix.

int ***SUNMatrix_cuSparse_IndexValues**(*SUNMatrix* A)
This function returns a pointer to the index value array for the sparse SUNMatrix – for the CSR format this is an array of 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.

int ***SUNMatrix_cuSparse_IndexPointers**(*SUNMatrix* A)
This function returns a pointer to the index pointer array for the sparse SUNMatrix – 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.

int **SUNMatrix_cuSparse_NumBlocks**(*SUNMatrix* A)
This function returns the number of matrix blocks.

int **SUNMatrix_cuSparse_BlockRows**(*SUNMatrix* A)
This function returns the number of rows in a matrix block.

int **SUNMatrix_cuSparse_BlockColumns**(*SUNMatrix* A)
This function returns the number of columns in a matrix block.

int **SUNMatrix_cuSparse_BlockNNZ**(*SUNMatrix* A)
This function returns the number of nonzeros in each matrix block.

realtype ***SUNMatrix_cuSparse_BlockData**(*SUNMatrix* A, int blockidx)
This function returns a pointer to the location in the data array where the data for the block, blockidx, begins. Thus, blockidx must be less than SUNMatrix_cuSparse_NumBlocks(A). The first block in the SUNMatrix is index 0, the second block is index 1, and so on.

cusparseMatDescr_t **SUNMatrix_cuSparse_MatDescr**(*SUNMatrix* A)
This function returns the cusparseMatDescr_t object associated with the matrix.

int **SUNMatrix_cuSparse_CopyToDevice**(*SUNMatrix* A, *realtype* *h_data, int *h_idxptrs, int *h_idxvals)
This functions 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.

The function returns SUNMAT_SUCCESS if the copy operation(s) were successful, or a nonzero error code otherwise.

int **SUNMatrix_cuSparse_CopyFromDevice**(*SUNMatrix* A, *realtype* *h_data, int *h_idxptrs, int *h_idxvals)
This functions 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. Otherwise:

- The h_data array must be at least SUNMatrix_cuSparse_NNZ(A)*sizeof(realtype) bytes.
- The h_idxptrs array must be at least (SUNMatrix_cuSparse_BlockDim(A)+1)*sizeof(int) bytes.
- The h_idxvals array must be at least (SUNMatrix_cuSparse_BlockNNZ(A))*sizeof(int) bytes.

The function returns SUNMAT_SUCCESS if the copy operation(s) were successful, or a nonzero error code otherwise.

int **SUNMatrix_cuSparse_SetFixedPattern**(*SUNMatrix* A, *booleantype* yesno)
This function changes the behavior of the the SUNMatZero operation on the 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.

int **SUNMatrix_cuSparse_SetKernelExecPolicy**(*SUNMatrix* A, SUNCudaExecPolicy *exec_policy)
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 §6.10.2 for more information about the *SUNCudaExecPolicy* class.

7.7.3 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.

Warning: When using the SUNMATRIX_CUSPARSE module with a SUNDIALS package (e.g. ARKODE), 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.

7.8 The SUNMATRIX_SPARSE Module

The sparse implementation of the `SUNMatrix` module, 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 in a sparse matrix is shown in Fig. 7.2. 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 at construction based the input choice 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 SUNMATRIX_SPARSE content structure 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 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 as a CSC matrix 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 be included when using this module is `sunmatrix/sunmatrix_sparse.h`.

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(A)
: This macro gives access to the contents of the sparse `SUNMatrix` *A*.

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

  Implementation:

  ```
  #define SM_CONTENT_S(A)   ( (SUNMatrixContent_Sparse)(A->content) )
  ```

SM_ROWS_S(A)
: Access the number of rows in the sparse `SUNMatrix` *A*.

  This may be used either to retrieve or to set the value. 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_ROWS_S(A) = A_rows` sets the number of columns in *A* to equal `A_rows`.

  Implementation:

  ```
  #define SM_ROWS_S(A)   ( SM_CONTENT_S(A)->M )
  ```

SM_COLUMNS_S(A)
: Access the number of columns in the sparse `SUNMatrix` *A*. As with `SM_ROWS_S`, this may be used either to retrieve or to set the value.

  Implementation:

  ```
  #define SM_COLUMNS_S(A)   ( SM_CONTENT_S(A)->N )
  ```

SM_NNZ_S(A)
: Access the allocated number of nonzeros in the sparse `SUNMatrix` *A*. As with `SM_ROWS_S`, this may be used either to retrieve or to set the value.

  Implementation:

  ```
  #define SM_NNZ_S(A)   ( SM_CONTENT_S(A)->NNZ )
  ```

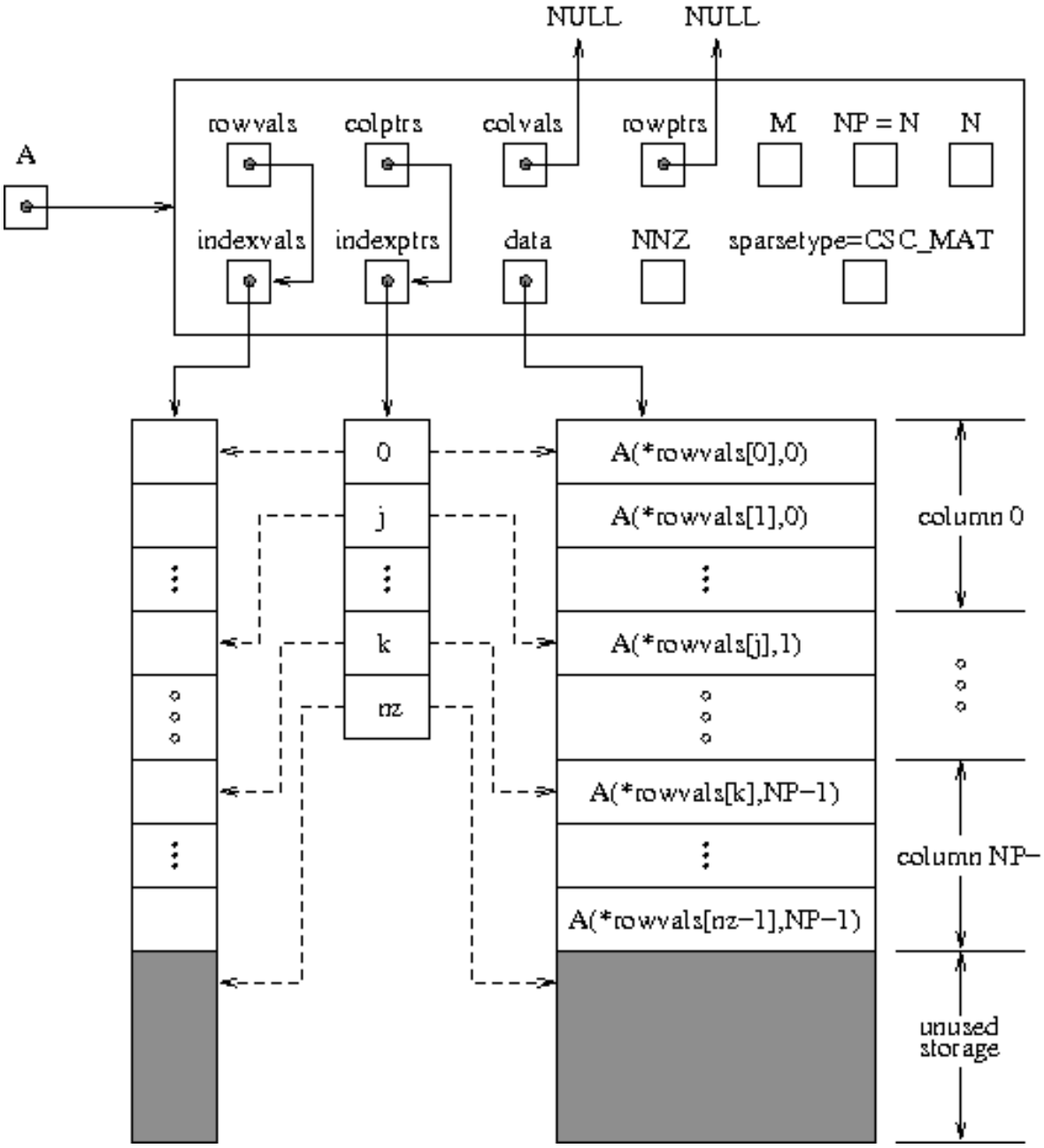

Fig. 7.2: Diagram of the storage for a compressed-sparse-column matrix of type SUNMATRIX_SPARSE: Here `A` is an $M \times N$ sparse CSC matrix with storage for up to `NNZ` nonzero entries (the allocated length of both `data` and `indexvals`). The entries in `indexvals` may assume values from `0` to `M-1`, 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.

SM_NP_S(A)
Access the number of index pointers NP in the sparse SUNMatrix *A*. As with SM_ROWS_S, this may be used either to retrieve or to set the value.

Implementation:

```
#define SM_NP_S(A)   ( SM_CONTENT_S(A)->NP )
```

SM_SPARSETYPE_S(A)
Access the sparsity type parameter in the sparse SUNMatrix *A*. As with SM_ROWS_S, this may be used either to retrieve or to set the value.

Implementation:

```
#define SM_SPARSETYPE_S(A)   ( SM_CONTENT_S(A)->sparsetype )
```

SM_DATA_S(A)
This macro gives access to the data pointer 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.

Implementation:

```
#define SM_DATA_S(A)   ( SM_CONTENT_S(A)->data )
```

SM_INDEXVALS_S(A)
This macro gives access to the indexvals pointer for the matrix entries.

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*.

Implementation:

```
#define SM_INDEXVALS_S(A)   ( SM_CONTENT_S(A)->indexvals )
```

SM_INDEXPTRS_S(A)
This macro gives access to the indexptrs pointer for the matrix entries.

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_INDEXPTRS_S(A)   ( SM_CONTENT_S(A)->indexptrs )
```

The SUNMATRIX_SPARSE module defines sparse implementations of all matrix operations listed in §7.2. Their names are obtained from those in that section by appending the suffix _Sparse (e.g. SUNMatCopy_Sparse). The module SUNMATRIX_SPARSE provides the following additional user-callable routines:

SUNMatrix **SUNSparseMatrix**(*sunindextype* M, *sunindextype* N, *sunindextype* NNZ, int sparsetype, *SUNContext* sunctx)
This constructor 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 choices are CSR_MAT or CSC_MAT).

SUNMatrix **SUNSparseFromDenseMatrix**(*SUNMatrix* A, *realtype* droptol, int sparsetype)
This constructor function creates a new sparse matrix from an existing SUNMATRIX_DENSE object by copying all values with magnitude larger than *droptol* into the sparse matrix structure.

Requirements:

- *A* must have type `SUNMATRIX_DENSE`
- *droptol* must be non-negative
- *sparsetype* must be either `CSC_MAT` or `CSR_MAT`

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

SUNMatrix **SUNSparseFromBandMatrix**(*SUNMatrix* A, *realtype* droptol, int sparsetype)
This constructor function creates a new sparse matrix from an existing SUNMATRIX_BAND object 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.

int **SUNSparseMatrix_Realloc**(*SUNMatrix* A)
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).

void **SUNSparseMatrix_Print**(*SUNMatrix* A, FILE *outfile)
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.

sunindextype **SUNSparseMatrix_Rows**(*SUNMatrix* A)
This function returns the number of rows in the sparse `SUNMatrix`.

sunindextype **SUNSparseMatrix_Columns**(*SUNMatrix* A)
This function returns the number of columns in the sparse `SUNMatrix`.

sunindextype **SUNSparseMatrix_NNZ**(*SUNMatrix* A)
This function returns the number of entries allocated for nonzero storage for the sparse `SUNMatrix`.

sunindextype **SUNSparseMatrix_NP**(*SUNMatrix* A)
This function returns the number of index pointers for the sparse `SUNMatrix` (the `indexptrs` array has `NP+1` entries).

int **SUNSparseMatrix_SparseType**(*SUNMatrix* A)
This function returns the storage type (`CSR_MAT` or `CSC_MAT`) for the sparse `SUNMatrix`.

realtype ***SUNSparseMatrix_Data**(*SUNMatrix* A)
This function returns a pointer to the data array for the sparse `SUNMatrix`.

sunindextype ***SUNSparseMatrix_IndexValues**(*SUNMatrix* A)
This function returns a pointer to index value array for the sparse `SUNMatrix` – for CSR format this is the column index for each nonzero entry, for CSC format this is the row index for each nonzero entry.

sunindextype ***SUNSparseMatrix_IndexPointers**(*SUNMatrix* A)

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.

Note: Within the `SUNMatMatvec_Sparse` routine, internal consistency checks are performed to ensure that the matrix is called with consistent `N_Vector` 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.

7.9 The SUNMATRIX_SLUNRLOC Module

The SUNMATRIX_SLUNRLOC module is an interface to the `SuperMatrix` structure provided by the SuperLU_DIST sparse matrix factorization and solver library written by X. Sherry Li and collaborators [17, 25, 26, 41]. It is designed to be used with the SuperLU_DIST `SUNLinearSolver` module discussed in §8.15. 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.

7.9.1 SUNMATRIX_SLUNRLOC Functions

The SUNMATRIX_SLUNRLOC module provides the following user-callable routines:

SUNMatrix **SUNMatrix_SLUNRloc**(SuperMatrix *Asuper, gridinfo_t *grid, *SUNContext* sunctx)
This constructor function creates and allocates memory for a SUNMATRIX_SLUNRLOC object. Its arguments are a fully-allocated SuperLU_DIST `SuperMatrix` with `Stype = SLU_NR_loc, Dtype = SLU_D, Mtype = SLU_GE` and an initialized SuperLU_DIST 2D process grid structure. It returns a `SUNMatrix` object if `Asuper` is compatible else it returns `NULL`.

void **SUNMatrix_SLUNRloc_Print**(*SUNMatrix* A, FILE *fp)
This function prints the underlying `SuperMatrix` content. It is useful for debugging. Its arguments are the `SUNMatrix` object and a `FILE` pointer to print to. It returns void.

SuperMatrix ***SUNMatrix_SLUNRloc_SuperMatrix**(*SUNMatrix* A)
This function returns the underlying `SuperMatrix` of `A`. Its only argument is the `SUNMatrix` object to access.

gridinfo_t ***SUNMatrix_SLUNRloc_ProcessGrid**(*SUNMatrix* A)
This function returns the SuperLU_DIST 2D process grid associated with `A`. Its only argument is the `SUNMatrix` object to access.

booleantype **SUNMatrix_SLUNRloc_OwnData**(*SUNMatrix* A)
This function returns true if the `SUNMatrix` object is responsible for freeing the underlying `SuperMatrix`, otherwise it returns false. Its only argument is the `SUNMatrix` object to access.

The SUNMATRIX_SLUNRLOC module also defines implementations of all generic `SUNMatrix` operations listed in §7.2:

- `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 $A = cA + B$, where A and B must have the same sparsity pattern
- `SUNMatScaleAddI_SLUNRloc` – performs $A = cA + I$, where 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`

7.10 SUNMATRIX Examples

There are `SUNMatrix` examples that may be installed for each implementation, that make use of the functions in `test_sunmatrix.c`. These example functions show simple usage of the `SUNMatrix` family of functions. The inputs to the examples depend on the matrix type, and are output to `stdout` if the example is run without the appropriate number of command-line arguments.

The following is a list of the example functions in `test_sunmatrix.c`:

- `Test_SUNMatGetID`: Verifies the returned matrix ID against the value that should be returned.

- `Test_SUNMatClone`: Creates clone of an existing matrix, copies the data, and checks that their values match.
- `Test_SUNMatZero`: Zeros out an existing matrix and checks that each entry equals 0.0.
- `Test_SUNMatCopy`: Clones an input matrix, copies its data to a clone, and verifies that all values match.
- `Test_SUNMatScaleAdd`: Given an input matrix A and an input identity matrix I, this test clones and copies A to a new matrix B, computes $B = -B + B$, and verifies that the resulting matrix entries equal 0. Additionally, if the matrix is square, this test clones and copies A to a new matrix D, clones and copies I to a new matrix C, computes $D = D + I$ and $C = C + A$ using *SUNMatScaleAdd()*, and then verifies that $C = D$.
- `Test_SUNMatScaleAddI`: Given an input matrix A and an input identity matrix I, this clones and copies I to a new matrix B, computes $B = -B + I$ using *SUNMatScaleAddI()*, and verifies that the resulting matrix entries equal 0.
- `Test_SUNMatMatvecSetup`: verifies that *SUNMatMatvecSetup()* can be called.
- `Test_SUNMatMatvec` Given an input matrix A and input vectors x and y such that $y = Ax$, this test has different behavior depending on whether A is square. If it is square, it clones and copies A to a new matrix B, computes $B = 3B + I$ using *SUNMatScaleAddI()*, clones y to new vectors w and z, computes $z = Bx$ using *SUNMatMatvec()*, computes $w = 3y + x$ using `N_VLinearSum`, and verifies that $w == z$. If A is not square, it just clones y to a new vector z, *computes :math: `z=Ax` using* *SUNMatMatvec()*, and verifies that $y = z$.
- `Test_SUNMatSpace`: verifies that *SUNMatSpace()* can be called, and outputs the results to `stdout`.

7.11 SUNMatrix functions used by KINSOL

In Table 7.2, 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 and KINBBDPRE preconditioner modules. 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 7.2: List of matrix functions usage by KINSOL code modules

	KINLS	KINBBDPRE
SUNMatGetID()	x	
SUNMatDestroy()		x
SUNMatZero()	x	x
SUNMatSpace()		†

The matrix functions listed 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 §7.1 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.

Chapter 8

Linear Algebraic Solvers

For problems that require the solution of linear systems of equations, the SUNDIALS packages operate using generic linear solver modules defined through the *SUNLinearSolver*, or "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 *N_Vector*, and optionally *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 `N_Vector` and/or `SUNMatrix` modules.

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

$$\tilde{A}\tilde{x} = \tilde{b} \tag{8.1}$$

where

$$\begin{aligned} \tilde{A} &= S_1 P_1^{-1} A P_2^{-1} S_2^{-1}, \\ \tilde{b} &= S_1 P_1^{-1} b, \\ \tilde{x} &= S_2 P_2 x, \end{aligned} \tag{8.2}$$

and where

- P_1 is the left preconditioner,
- P_2 is the right preconditioner,
- S_1 is a diagonal matrix of scale factors for $P_1^{-1} b$,
- S_2 is a diagonal matrix of scale factors for $P_2 x$.

SUNDIALS solvers request that iterative linear solvers stop based on the 2-norm of the scaled preconditioned residual meeting a prescribed tolerance, i.e.,

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

When provided an iterative SUNLinSol implementation that does not support the scaling matrices S_1 and S_2, the SUNDIALS packages will adjust the value of tol accordingly (see the iterative linear tolerance section that follows for more details). In this case, they instead request that iterative linear solvers stop based on the criterion

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

We note that the corresponding adjustments to tol in this case may not be 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 `SUNLinearSolver` 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 §8.1.1 – §8.1.3. This is followed by the definition of functions supplied to a linear solver implementation in §8.1.4. The linear solver return codes are described in Table 8.1. The `SUNLinearSolver` type and the generic SUNLinSol module are defined in §8.1.6. §8.1.8 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 section that then follows describes the SUNLinSol functions required by this SUNDIALS package, and provides additional package specific details. Then the remaining sections of this chapter present the SUNLinSol modules provided with SUNDIALS.

8.1 The SUNLinearSolver API

The SUNLinSol API defines several linear solver operations that enable SUNDIALS packages to utilize this API. These functions can be divided into three categories. The first are the core linear solver functions. The second consist of "set" routines to supply the linear solver with functions provided by the SUNDIALS packages 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`.

8.1.1 SUNLinearSolver core functions

The core linear solver functions consist of two **required** functions: *SUNLinSolGetType()* returns the linear solver type, and *SUNLinSolSolve()* solves the linear system $Ax = b$.

The remaining **optional** functions return the solver ID (*SUNLinSolGetID()*), initialize the linear solver object once all solver-specific options have been set (*SUNLinSolInitialize()*), set up the linear solver object to utilize an updated matrix A (*SUNLinSolSetup()*), and destroy a linear solver object (*SUNLinSolFree()*).

SUNLinearSolver_Type **SUNLinSolGetType**(*SUNLinearSolver* LS)

Returns the type identifier for the linear solver *LS*.

Return value:

- `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.

Usage:

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

Note: See §8.1.8.1 for more information on intended use cases corresponding to the linear solver type.

SUNLinearSolver_ID **SUNLinSolGetID**(*SUNLinearSolver* LS)

Returns a non-negative linear solver identifier (of type int) for the linear solver *LS*.

Return value:

Non-negative linear solver identifier (of type int), defined by the enumeration SUNLinearSolver_ID, with values shown in Table 8.2 and defined in the sundials_linearsolver.h header file.

Usage:

```
id = SUNLinSolGetID(LS);
```

Note: It is recommended that a user-supplied SUNLinearSolver return the SUNLINEARSOLVER_CUSTOM identifier.

---

int **SUNLinSolInitialize**(*SUNLinearSolver* LS)

Performs linear solver initialization (assuming that all solver-specific options have been set).

Return value:

Zero for a successful call, and a negative value for a failure. Ideally, this should return one of the generic error codes listed in Table 8.1.

Usage:

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

int **SUNLinSolSetup**(*SUNLinearSolver* LS, *SUNMatrix* A)

Performs any linear solver setup needed, based on an updated system SUNMatrix *A*. This may be called frequently (e.g., with a full Newton method) or infrequently (for a modified Newton method), based on the type of integrator and/or nonlinear solver requesting the solves.

Return value:

Zero for a successful call, a positive value for a recoverable failure, and a negative value for an unrecoverable failure. Ideally this should return one of the generic error codes listed in Table 8.1.

Usage:

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

int **SUNLinSolSolve**(*SUNLinearSolver* LS, *SUNMatrix* A, *N_Vector* x, *N_Vector* b, *realtype* tol)

This *required* function solves a linear system $Ax = b$.

Arguments:

- *LS* – a SUNLinSol object.
- *A* – a `SUNMatrix` object.
- *x* – an `N_Vector` object containing the initial guess for the solution of the linear system on input, and the solution to the linear system upon return.
- *b* – an `N_Vector` object containing the linear system right-hand side.
- *tol* – the desired linear solver tolerance.

Return value:

Zero for a successful call, a positive value for a recoverable failure, and a negative value for an unrecoverable failure. Ideally this should return one of the generic error codes listed in Table 8.1.

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 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 function, the solver will call interface routines from the relevant SUNDIALS package to directly form the linear system matrix A, and then solve $Ax = b$ before returning with the solution x.

Usage:

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

int **SUNLinSolFree**(*SUNLinearSolver* LS)

Frees memory allocated by the linear solver.

Return value:

Zero for a successful call, and a negative value for a failure. Ideally, this should return one of the generic error codes listed in Table 8.1.

Usage:

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

8.1.2 SUNLinearSolver "set" functions

The following functions supply linear solver modules with functions defined by the SUNDIALS packages and 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.

int **SUNLinSolSetATimes**(*SUNLinearSolver* LS, void *A_data, *SUNATimesFn* ATimes)
Required for matrix-free linear solvers (otherwise optional).

Provides a *SUNATimesFn* function pointer, as well as a `void*` pointer to a data structure used by this routine, to the linear solver object *LS*. SUNDIALS packages 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.

Return value:

Zero for a successful call, and a negative value for a failure. Ideally, this should return one of the generic error codes listed in Table 8.1.

Usage:

```
retval = SUNLinSolSetATimes(LS, A_data, ATimes);
```

int **SUNLinSolSetPreconditioner**(*SUNLinearSolver* LS, void *P_data, *SUNPSetupFn* Pset, *SUNPSolveFn* Psol)
This *optional* routine provides *SUNPSetupFn* and *SUNPSolveFn* function pointers that implement the preconditioner solves P_1^{-1} and P_2^{-1} from (8.2). This routine is called by a SUNDIALS package, which provides translation between the generic *Pset* and *Psol* calls and the package- or user-supplied routines.

Return value:

Zero for a successful call, and a negative value for a failure. Ideally, this should return one of the generic error codes listed in Table 8.1.

Usage:

```
retval = SUNLinSolSetPreconditioner(LS, Pdata, Pset, Psol);
```

int **SUNLinSolSetScalingVectors**(*SUNLinearSolver* LS, *N_Vector* s1, *N_Vector* s2)
This *optional* routine provides left/right scaling vectors for the linear system solve. Here, *s1* and *s2* are `N_Vectors` of positive scale factors containing the diagonal of the matrices S_1 and S_2 from (8.2), respectively. Neither vector needs to be tested for positivity, and a `NULL` argument for either indicates that the corresponding scaling matrix is the identity.

Return value:

Zero for a successful call, and a negative value for a failure. Ideally, this should return one of the generic error codes listed in Table 8.1.

Usage:

```
retval = SUNLinSolSetScalingVectors(LS, s1, s2);
```

int **SUNLinSolSetZeroGuess**(*SUNLinearSolver* LS, *booleantype* onoff)
This *optional* routine indicates if the upcoming `SUNlinSolSolve()` call will be made with a zero initial guess (`SUNTRUE`) or a non-zero initial guess (`SUNFALSE`).

Return value:

Zero for a successful call, and a negative value for a failure. Ideally, this should return one of the generic error codes listed in Table 8.1.

Usage:

```
retval = SUNLinSolSetZeroGuess(LS, onoff);
```

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()*.

8.1.3 SUNLinearSolver "get" functions

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

int **SUNLinSolNumIters**(*SUNLinearSolver* LS)
This *optional* routine should return the number of linear iterations performed in the most-recent "solve" call.

Usage:

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

realtype **SUNLinSolResNorm**(*SUNLinearSolver* LS)
This *optional* routine should return the final residual norm from the most-recent "solve" call.

Usage:

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

N_Vector **SUNLinSolResid**(*SUNLinearSolver* LS)
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 `N_Vector` containing the preconditioned initial residual vector.

Usage:

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

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.

sunindextype **SUNLinSolLastFlag**(*SUNLinearSolver* LS)
This *optional* routine should return the last error flag encountered within the linear solver. Although not called by the SUNDIALS packages directly, this may be called by the user to investigate linear solver issues after a failed solve.

Usage:

```
lflag = SUNLinLastFlag(LS);
```

int **SUNLinSolSpace**(*SUNLinearSolver* LS, long int *lenrwLS, long int *leniwLS)
This *optional* routine should return the storage requirements for the linear solver *LS*:

- *lrw* is a `long int` containing the number of realtype words
- *liw* is a `long int` containing the number of integer words.

The return value is an integer flag denoting success/failure of the operation.

This function is advisory only, for use by users to help determine their total space requirements.

Usage:

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

8.1.4 Functions provided by SUNDIALS packages

To interface with 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 linear solver API. The function types for these routines are defined in the header file `sundials/sundials_iterative.h`, and are described below.

typedef int (***SUNATimesFn**)(void *A_data, *N_Vector* v, *N_Vector* z)

Computes the action of a matrix on a vector, performing the operation $z \leftarrow Av$. Memory for z will already be allocated prior to calling this function. The parameter *A_data* is a pointer to any information about A which the function needs in order to do its job. The vector v should be left unchanged.

Return value:

Zero for a successful call, and non-zero upon failure.

typedef int (***SUNPSetupFn**)(void *P_data)

Sets up any requisite problem data in preparation for calls to the corresponding *SUNPSolveFn*.

Return value:

Zero for a successful call, and non-zero upon failure.

typedef int (***SUNPSolveFn**)(void *P_data, *N_Vector* r, *N_Vector* z, *realtype* tol, int lr)

Solves the preconditioner equation $Pz = r$ for the vector z. Memory for z will 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 *SUNPSetupFn*). The parameter *lr* is input, and indicates whether P is to be taken as the left or right preconditioner: *lr* = 1 for left and *lr* = 2 for right. If preconditioning is on one side only, *lr* can be ignored. If the preconditioner is iterative, then it should strive to solve the preconditioner equation so that

$$\|Pz - r\|_{\text{wrms}} < tol$$

where the error weight vector for the WRMS norm may be accessed from the main package memory structure. The vector *r* should not be modified by the *SUNPSolveFn*.

Return value:

Zero for a successful call, a negative value for an unrecoverable failure condition, or a positive value for a recoverable failure condition (thus the calling routine may reattempt the solution after updating preconditioner data).

8.1.5 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, listed in Table 8.1. These adhere to a common pattern:

- 0 indicates success
- a positive value corresponds to a recoverable failure, and
- a negative value indicates a non-recoverable failure.

Aside from this pattern, the actual values of each error code provide additional information to the user in case of a linear solver failure.

Table 8.1: SUNLinSol error codes

Error code	Value	Meaning
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 (SPGMR/SPFGMR)
SUNLS_QRSOL_FAIL	-811	a singular R matrix was encountered in a QR factorization (SPGMR/SPFGMR)
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
SUNLS_QRFACT_FAIL	807	a singular matrix was encountered during a QR factorization (SPGMR/SPFGMR)
SUNLS_LUFACT_FAIL	808	a singular matrix was encountered during a LU factorization

8.1.6 The generic SUNLinearSolver module

SUNDIALS packages interact with specific SUNLinSol implementations through the generic SUNLinearSolver abstract base class. The `SUNLinearSolver` type is a pointer to a structure containing an implementation-dependent *content* field, and an *ops* field, and is defined as

typedef struct _generic_SUNLinearSolver ***SUNLinearSolver**

and the generic structure is defined as

```
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*, SUNATimesFn);
  int                  (*setpreconditioner)(SUNLinearSolver, void*,
                                            SUNPSetupFn, SUNPSolveFn);
  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);
  sunindextype         (*lastflag)(SUNLinearSolver);
  int                  (*space)(SUNLinearSolver, long int*, long int*);
  N_Vector             (*resid)(SUNLinearSolver);
  int                  (*free)(SUNLinearSolver);
};
```

The generic SUNLinSol class defines and implements the linear solver operations defined in §8.1.1 – §8.1.3. 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 `SUNLinearSolver` base class, namely *SUNLinSolInitialize()*, that initializes a `SUNLinearSolver` object for use after it has been created and configured, and returns a flag denoting a successful or failed operation:

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

8.1.7 Compatibility of SUNLinearSolver modules

Not all `SUNLinearSolver` implementations are compatible with all `SUNMatrix` and `N_Vector` implementations provided in SUNDIALS. More specifically, all of the SUNDIALS iterative linear solvers (*SPGMR*, *SPFGMR*, *SPBCGS*, *SPTFQMR*, and *PCG*) are compatible with all of the SUNDIALS `N_Vector` modules, but the matrix-based direct SUNLinSol modules are specifically designed to work with distinct `SUNMatrix` and `N_Vector` modules. In the list below, we summarize the compatibility of each matrix-based `SUNLinearSolver` module with the various `SUNMatrix` and `N_Vector` modules. For a more thorough discussion of these compatibilities, we defer to the documentation for each individual SUNLinSol module in the sections that follow.

- *Dense*
  - `SUNMatrix`: *Dense* or user-supplied
  - `N_Vector`: *Serial*, *OpenMP*, *Pthreads*, or user-supplied
- *LapackDense*
  - `SUNMatrix`: *Dense* or user-supplied
  - `N_Vector`: *Serial*, *OpenMP*, *Pthreads*, or user-supplied
- *Band*
  - `SUNMatrix`: *Band* or user-supplied
  - `N_Vector`: *Serial*, *OpenMP*, *Pthreads*, or user-supplied
- *LapackBand*
  - `SUNMatrix`: *Band* or user-supplied
  - `N_Vector`: *Serial*, *OpenMP*, *Pthreads*, or user-supplied
- *KLU*
  - `SUNMatrix`: *Sparse* or user-supplied
  - `N_Vector`: *Serial*, *OpenMP*, *Pthreads*, or user-supplied
- *SuperLU_MT*
  - `SUNMatrix`: *Sparse* or user-supplied
  - `N_Vector`: *Serial*, *OpenMP*, *Pthreads*, or user-supplied
- *SuperLU_Dist*
  - `SUNMatrix`: *SLUNRLOC* or user-supplied
  - `N_Vector`: *Serial*, *OpenMP*, *Pthreads*, *Parallel*, **hypre**, *PETSc*, or user-supplied
- *Magma Dense*
  - `SUNMatrix`: *Magma Dense* or user-supplied
  - `N_Vector`: *HIP*, *RAJA*, or user-supplied
- *OneMKL Dense*
  - `SUNMatrix`: *One MKL Dense* or user-supplied
  - `N_Vector`: *SYCL*, *RAJA*, or user-supplied
- *cuSolverSp batchQR*
  - `SUNMatrix`: *cuSparse* or user-supplied
  - `N_Vector`: *CUDA*, *RAJA*, or user-supplied

8.1.8 Implementing a custom SUNLinearSolver module

A particular implementation of the SUNLinearSolver module must:

- Specify the *content* field of the SUNLinSol module.
- Define and implement the required linear solver operations.

  Note: 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 whether the associated functionality is supported.

To aid in the creation of custom SUNLinearSolver modules the generic SUNLinearSolver module provides the utility function *SUNLinSolNewEmpty()*. When used in custom SUNLinearSolver constructors this function will ease the introduction of any new optional linear solver operations to the SUNLinearSolver API by ensuring that only required operations need to be set.

SUNLinearSolver **SUNLinSolNewEmpty**()
This function allocates a new generic SUNLinearSolver object and initializes its content pointer and the function pointers in the operations structure to NULL.

Return value:

If successful, this function returns a SUNLinearSolver object. If an error occurs when allocating the object, then this routine will return NULL.

void **SUNLinSolFreeEmpty**(*SUNLinearSolver* LS)
This routine frees the generic SUNLinearSolver 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* – a SUNLinearSolver object

Additionally, a SUNLinearSolver 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 for a particular problem.
- Provide additional user-callable "get" routines acting on the SUNLinearSolver object, e.g., for returning various solve statistics.

Each SUNLinSol implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 8.2. It is recommended that a user-supplied SUNLinSol implementation use the SUNLINEARSOLVER_CUSTOM identifier.

Table 8.2: Identifiers associated with `SUNLinearSolver` modules supplied with SUNDIALS

SUNLinSol ID	Linear solver type	ID Value
SUNLINEARSOLVER_BAND	Banded direct linear solver (internal)	0
SUNLINEARSOLVER_DENSE	Dense direct linear solver (internal)	1
SUNLINEARSOLVER_KLU	Sparse direct linear solver (KLU)	2
SUNLINEARSOLVER_LAPACKBAND	Banded direct linear solver (LAPACK)	3
SUNLINEARSOLVER_LAPACKDENSE	Dense direct linear solver (LAPACK)	4
SUNLINEARSOLVER_PCG	Preconditioned conjugate gradient iterative solver	5
SUNLINEARSOLVER_SPBCGS	Scaled-preconditioned BiCGStab iterative solver	6
SUNLINEARSOLVER_SPFGMR	Scaled-preconditioned FGMRES iterative solver	7
SUNLINEARSOLVER_SPGMR	Scaled-preconditioned GMRES iterative solver	8
SUNLINEARSOLVER_SPTFQMR	Scaled-preconditioned TFQMR iterative solver	9
SUNLINEARSOLVER_SUPERLUDIST	Parallel sparse direct linear solver (SuperLU_Dist)	10
SUNLINEARSOLVER_SUPERLUMT	Threaded sparse direct linear solver (SuperLU_MT)	11
SUNLINEARSOLVER_CUSOLVERSP_BATCHQR	Sparse direct linear solver (CUDA)	12
SUNLINEARSOLVER_MAGMADENSE	Dense or block-dense direct linear solver (MAGMA)	13
SUNLINEARSOLVER_ONEMKLDENSE	Dense or block-dense direct linear solver (OneMKL)	14
SUNLINEARSOLVER_CUSTOM	User-provided custom linear solver	15

8.1.8.1 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. Many external solvers provide routines with similar functionality and thus may require minimal effort to wrap within custom SUNMATRIX and SUNLinSol implementations. As SUNDIALS packages utilize generic SUNLinSol modules they may naturally leverage user-supplied `SUNLinearSolver` implementations, thus there exist a wide range of possible linear solver combinations. Some intended use cases for both the SUNDIALS-provided and user-supplied SUNLinSol modules are discussd in the sections below.

Direct linear solvers

Direct linear solver modules require a matrix and compute an "exact" solution to the linear system *defined by the matrix*. SUNDIALS packages strive to amortize the high cost of matrix construction by reusing matrix information for multiple nonlinear iterations or time steps. As a result, each package's linear solver interface recomputes matrix 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 §7 and §8. *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 instead compute an inexact solution to the linear system *defined by the package-supplied* `ATimes` *routine*. SUNDIALS supplies multiple scaled, preconditioned iterative SUNLinSol modules that support scaling, allowing packages to handle non-dimensionalization, and users to define variables and equations as natural in their applications. However, for linear solvers that do not support left/right scaling, SUNDIALS packages must instead adjust the tolerance supplied to the linear solver to compensate (see the iterative linear tolerance section that follows for more details) – this strategy may be non-optimal since it cannot handle situations where the magnitudes of different solution components or equations vary dramatically within a single application.

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 §8. *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 resued across multiple solves to amortize the cost of matrix construction. As in the direct linear solver case, only thin SUNMATRIX and SUNLinSol wrappers for the underlying matrix and linear solver structures need to be created to utilize such a 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 *hypre* library; this may be used as a template for other customized implementations (see `examples/arkode/CXX_parhyp/ark_heat2D_hypre.cpp`).

Matrix-based iterative linear solvers (current A)

For users who wish to utilize a matrix-based iterative linear solver 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 scaled, preconditioned iterative linear solver implementations (*SUNLinSol_SPGMR()*, *SUNLinSol_SPFGMR()*, *SUNLinSol_SPBCGS()*, *SUNLinSol_SPTFQMR()*, or *SUNLinSol_PCG()*) as the outer solver. The creation and storage of the preconditioner matrix, and interfacing with the corresponding matrix-based linear solver, can be handled through a package's preconditioner "setup" and "solve" functionality without creating SUNMATRIX and SUNLinSol implementations. This usage mode is recommended primarily because the SUNDIALS-provided modules support variable and equation 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 `SUNATimesFn` routine.

Application-specific linear solvers with embedded matrix structure

Many applications can exploit additional linear system structure arising from 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. In many of the above 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, that only needs to implement the required *SUNLinSolGetType()* and *SUNLinSolSolve()* operations. Within *SUNLinSolSolve()*, the linear solver implementation should call package-specific interface routines (e.g., `ARKStepGetNonlinearSystemData`, `CVodeGetNonlinearSystemData`, `IDAGetNonlinearSystemData`, `ARKStepGetCurrentGamma`, `CVodeGetCurrentGamma`, `IDAGetCurrentCj`, or `MRIStepGetCurrentGamma`) 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 SUNLinearSolver object with the relevant SUNDIALS package `SetLinearSolver` routine, the input *SUNMatrix* `A` should be set to `NULL`.

For templates of such user-provided "matrix-embedded" SUNLinSol implementations, see the SUNDIALS examples `ark_analytic_mels.c`, `cvAnalytic_mels.c`, `cvsAnalytic_mels.c`, `idaAnalytic_mels.c`, and `idasAnalytic_mels.c`.

8.2 KINSOL SUNLinearSolver interface

Table 8.3 below lists the `SUNLinearSolver` 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 "x" to indicate that they are required, or with "†" to indicate that they are only called if they are non-`NULL` in the `SUNLinearSolver` 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 `SUNLinearSolver` module, then KINLS will consider all solves as requiring zero iterations.
2. Although *SUNLinSolResNorm()* is optional, if it is not implemented by the `SUNLinearSolver` 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 8.3: List of linear solver function usage in the KINLS interface

SUNLinSolGetType	x	x	x
SUNLinSolSetATimes	†	x	†
SUNLinSolSetPreconditioner	†	†	†
SUNLinSolSetScalingVectors	†	†	†
SUNLinSolInitialize	x	x	x
SUNLinSolSetup	x	x	x
SUNLinSolSolve	x	x	x
[1]SUNLinSolNumIters		†	†
[2]SUNLinSolResNorm		†	†
[3]SUNLinSolLastFlag			
[4]SUNLinSolFree			
SUNLinSolSpace	†	†	†

Since there are a wide range of potential SUNLinearSolver use cases, the following subsections describe some details of the KINLS interface, in the case that interested users wish to develop custom SUNLinearSolver modules.

8.2.1 Lagged matrix information

If the SUNLinearSolver object self-identifies as having type SUNLINEARSOLVER_DIRECT or SUNLINEARSOLVER_MATRIX_ITERATIVE, then the SUNLinearSolver object solves a linear system *defined* by a SUNMatrix object. As a result, KINSOL can perform its optional residual monitoring scheme, described in §2.8.

8.2.2 Iterative linear solver tolerance

If the SUNLinearSolver object self-identifies as having type SUNLINEARSOLVER_ITERATIVE or SUNLINEARSOLVER_MATRIX_ITERATIVE then KINLS will adjust the linear solver tolerance delta as described in §2.9 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 §2.4. 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 §2.4) should satisfy the assumption

$$(D_F)_{i,i} \approx D_{F,mean}, \quad \text{for} \quad i = 0, \ldots, n-1.$$

2. The SUNLinearSolver 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 con-

verted as follows (using the notation from equations (8.1) – (8.2):

$$\begin{aligned}
& \|\tilde{b} - \tilde{A}\tilde{x}\|_2 < \text{tol} \\
\Leftrightarrow \quad & \|D_F P_1^{-1} b - D_F P_1^{-1} A x\|_2 < \text{tol} \\
\Leftrightarrow \quad & \sum_{i=0}^{n-1} \left[(D_F)_{i,i} \left(P_1^{-1}(b - Ax) \right)_i \right]^2 < \text{tol}^2 \\
\Leftrightarrow \quad & D_{F,mean}^2 \sum_{i=0}^{n-1} \left[\left(P_1^{-1}(b - Ax) \right)_i \right]^2 < \text{tol}^2 \\
\Leftrightarrow \quad & \sum_{i=0}^{n-1} \left[\left(P_1^{-1}(b - Ax) \right)_i \right]^2 < \left(\frac{\text{tol}}{D_{F,mean}} \right)^2 \\
\Leftrightarrow \quad & \|P_1^{-1}(b - Ax)\|_2 < \frac{\text{tol}}{D_{F,mean}}
\end{aligned}$$

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`$= \text{tol}/D_{F,mean}$ is supplied to the `SUNLinearSolver` object.

8.2.3 Matrix-embedded solver incompatibility

At present, KINLS is incompatible with `SUNLinearSolver` 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.

8.3 The SUNLinSol_Band Module

The SUNLinSol_Band implementation of the `SUNLinearSolver` class is designed to be used with the corresponding *SUNMATRIX_BAND* matrix type, and one of the serial or shared-memory `N_Vector` implementations (NVECTOR_SERIAL, NVECTOR_OPENMP or NVECTOR_PTHREADS).

8.3.1 SUNLinSol_Band Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_band.h`. The SUNLinSol_Band module is accessible from all SUNDIALS packages *without* linking to the `libsundials_sunlinsolband` module library.

The SUNLinSol_Band module provides the following user-callable constructor routine:

SUNLinearSolver **SUNLinSol_Band**(*N_Vector* y, *SUNMatrix* A, *SUNContext* sunctx)
This function creates and allocates memory for a band `SUNLinearSolver`.

Arguments:

- *y* – vector used to determine the linear system size
- *A* – matrix used to assess compatibility
- *sunctx* – the *SUNContext* object (see §4.1)

Return value: New SUNLinSol_Band object, or `NULL` if either `A` or `y` are incompatible.

Notes: This routine will perform consistency checks to ensure that it is called with consistent `N_Vector` 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.

For backwards compatibility, we also provide the following wrapper function:

SUNLinearSolver **SUNBandLinearSolver**(*N_Vector* y, *SUNMatrix* A)
Wrapper function for *SUNLinSol_Band()*, with identical input and output arguments.

8.3.2 SUNLinSol_Band Description

The SUNLinSol_Band module defines the *content* field of a `SUNLinearSolver` to be 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.

This solver is constructed to perform the following operations:

- The "setup" call performs an LU factorization with partial (row) pivoting, $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix with 1's on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX_BAND object A, with pivoting information encoding P stored in the `pivots` array.
- The "solve" call performs pivoting and forward and backward substitution using the stored `pivots` array and the LU factors held in the SUNMATRIX_BAND object.
- A must be allocated to accommodate the increase in upper bandwidth that occurs during factorization. More precisely, if A is a band matrix with upper bandwidth `mu` and lower bandwidth `ml`, then the upper triangular factor U can have upper bandwidth as big as `smu = MIN(N-1,mu+ml)`. The lower triangular factor L has lower bandwidth `ml`.

The SUNLinSol_Band module defines band implementations of all "direct" linear solver operations listed in §8.1:

- `SUNLinSolGetType_Band`
- `SUNLinSolInitialize_Band` – this does nothing, since all consistency checks are performed at solver creation.
- `SUNLinSolSetup_Band` – this performs the LU factorization.
- `SUNLinSolSolve_Band` – this uses the LU factors and `pivots` array to perform the solve.
- `SUNLinSolLastFlag_Band`

- SUNLinSolSpace_Band – this only returns information for the storage *within* the solver object, i.e. storage for N, last_flag, and pivots.
- SUNLinSolFree_Band

8.4 The SUNLinSol_Dense Module

The SUNLinSol_Dense implementation of the SUNLinearSolver class is designed to be used with the corresponding SUNMATRIX_DENSE matrix type, and one of the serial or shared-memory N_Vector implementations (NVECTOR_SERIAL, NVECTOR_OPENMP or NVECTOR_PTHREADS).

8.4.1 SUNLinSol_Dense Usage

The header file to be included when using this module is sunlinsol/sunlinsol_dense.h. The SUNLinSol_Dense module is accessible from all SUNDIALS solvers *without* linking to the libsundials_sunlinsoldense module library.

The module SUNLinSol_Dense provides the following user-callable constructor routine:

SUNLinearSolver **SUNLinSol_Dense**(*N_Vector* y, *SUNMatrix* A, *SUNContext* sunctx)
This function creates and allocates memory for a dense SUNLinearSolver.

Arguments:

- *y* – vector used to determine the linear system size.
- *A* – matrix used to assess compatibility.
- *sunctx* – the *SUNContext* object (see §4.1)

Return value: New SUNLinSol_Dense object, or NULL if either A or y are incompatible.

Notes: This routine will perform consistency checks to ensure that it is called with consistent N_Vector 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.

For backwards compatibility, we also provide the following wrapper function:

SUNLinearSolver **SUNDenseLinearSolver**(*N_Vector* y, *SUNMatrix* A)
Wrapper function for *SUNLinSol_Dense()*, with identical input and output arguments

8.4.2 SUNLinSol_Dense Description

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

```
struct _SUNLinearSolverContent_Dense {
  sunindextype N;
  sunindextype *pivots;
  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.

This solver is constructed to perform the following operations:

- The "setup" call performs an LU factorization with partial (row) pivoting ($\mathcal{O}(N^3)$ cost), $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix with 1's on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX_DENSE object A, with pivoting information encoding P stored in the `pivots` array.
- The "solve" call performs pivoting and forward and backward substitution using the stored `pivots` array and the LU factors held in the SUNMATRIX_DENSE object ($\mathcal{O}(N^2)$ cost).

The SUNLinSol_Dense module defines dense implementations of all "direct" linear solver operations listed in §8.1:

- `SUNLinSolGetType_Dense`
- `SUNLinSolInitialize_Dense` – this does nothing, since all consistency checks are performed at solver creation.
- `SUNLinSolSetup_Dense` – this performs the LU factorization.
- `SUNLinSolSolve_Dense` – this uses the LU factors and `pivots` array to perform the solve.
- `SUNLinSolLastFlag_Dense`
- `SUNLinSolSpace_Dense` – this only returns information for the storage *within* the solver object, i.e. storage for `N`, `last_flag`, and `pivots`.
- `SUNLinSolFree_Dense`

8.5 The SUNLinSol_KLU Module

The SUNLinSol_KLU implementation of the `SUNLinearSolver` class is designed to be used with the corresponding SUNMATRIX_SPARSE matrix type, and one of the serial or shared-memory `N_Vector` implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, or NVECTOR_PTHREADS).

8.5.1 SUNLinSol_KLU Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_klu.h`. The installed module library to link to is `libsundials_sunlinsolklu` *.lib* where *.lib* is typically `.so` for shared libraries and `.a` for static libraries.

The module SUNLinSol_KLU provides the following additional user-callable routines:

SUNLinearSolver **SUNLinSol_KLU**(*N_Vector* y, *SUNMatrix* A, *SUNContext* sunctx)
This constructor function creates and allocates memory for a SUNLinSol_KLU object.

Arguments:

- *y* – vector used to determine the linear system size.
- *A* – matrix used to assess compatibility.
- *sunctx* – the `SUNContext` object (see §4.1)

Return value: New SUNLinSol_KLU object, or `NULL` if either `A` or `y` are incompatible.

Notes: This routine will perform consistency checks to ensure that it is called with consistent `N_Vector` 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.

int **SUNLinSol_KLUReInit**(*SUNLinearSolver* S, *SUNMatrix* A, *sunindextype* nnz, int reinit_type)

This function reinitializes memory and flags for a new factorization (symbolic and numeric) to be conducted at the next solver setup call. This routine is useful in the cases where the number of nonzeroes has changed or if the structure of the linear system has changed which would require a new symbolic (and numeric factorization).

Arguments:

- *S* – existing SUNLinSol_KLU object to reinitialize.
- *A* – sparse `SUNMatrix` matrix (with updated structure) to use for reinitialization.
- *nnz* – maximum number of nonzeros expected for Jacobian matrix.
- *reinit_type* – governs the level of reinitialization. The allowed values are:
  1. The Jacobian matrix will be destroyed and a new one will be allocated based on the `nnz` value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup.
  2. Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of `nnz` given in the sparse matrix provided to the original constructor routine (or the previous `SUNKLUReInit` call).

Return value:

- `SUNLS_SUCCESS` – reinitialization successful.
- `SUNLS_MEM_NULL` – either `S` or `A` are `NULL`.
- **`SUNLS_ILL_INPUT` – `A` does not have type `SUNMATRIX_SPARSE` or** `reinit_type` is invalid.
- `SUNLS_MEM_FAIL` reallocation of the sparse matrix failed.

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

int **SUNLinSol_KLUSetOrdering**(*SUNLinearSolver* S, int ordering_choice)

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

Arguments:

- *S* – existing SUNLinSol_KLU object to update.
- *ordering_choice* – type of ordering to use, options are:
  0. AMD,
  1. COLAMD, and
  2. the natural ordering.

  The default is 1 for COLAMD.

Return value:

- `SUNLS_SUCCESS` – ordering choice successfully updated.
- `SUNLS_MEM_NULL` – `S` is `NULL`.
- `SUNLS_ILL_INPUT` – `ordering_choice`.

sun_klu_symbolic ***SUNLinSol_KLUGetSymbolic**(*SUNLinearSolver* S)
This function returns a pointer to the KLU symbolic factorization stored in the SUNLinSol_KLU `content` structure.

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 compiled with 64-bit indices (`SUNDIALS_INDEX_SIZE=64`) this is mapped to the KLU type `klu_l_symbolic`.

sun_klu_numeric ***SUNLinSol_KLUGetNumeric**(*SUNLinearSolver* S)
This function returns a pointer to the KLU numeric factorization stored in the SUNLinSol_KLU `content` structure.

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`.

sun_klu_common ***SUNLinSol_KLUGetCommon**(*SUNLinearSolver* S)
This function returns a pointer to the KLU common structure stored in the SUNLinSol_KLU `content` structure.

When SUNDIALS is compiled with 32-bit indices (`SUNDIALS_INDEX_SIZE=32`), `sun_klu_common` is mapped to the KLU type `klu_common`; when SUNDIALS is compiled with 64-bit indices (`SUNDIALS_INDEX_SIZE=64`) this is mapped to the KLU type `klu_l_common`.

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

SUNLinearSolver **SUNKLU**(*N_Vector* y, *SUNMatrix* A)
Wrapper function for *SUNLinSol_KLU()*

int **SUNKLUReInit**(*SUNLinearSolver* S, *SUNMatrix* A, *sunindextype* nnz, int reinit_type)
Wrapper function for *SUNLinSol_KLUReInit()*

int **SUNKLUSetOrdering**(*SUNLinearSolver* S, int ordering_choice)
Wrapper function for *SUNLinSol_KLUSetOrdering()*

8.5.2 SUNLinSol_KLU Description

The SUNLinSol_KLU module defines the *content* field of a `SUNLinearSolver` to be 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).

The SUNLinSol_KLU module is a `SUNLinearSolver` wrapper for the KLU sparse matrix factorization and solver library written by Tim Davis and collaborators ([9, 37]). 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 §10.1.4 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 *Data Types* for details). Since the KLU library supports both 32-bit and 64-bit integers, this interface will be compiled for either of the available *`sunindextype`* options.

The KLU library has a symbolic factorization routine that computes the permutation of the linear system matrix to block triangular form and the permutations that will pre-order the diagonal blocks (the only ones that need to be factored) to reduce fill-in (using AMD, COLAMD, CHOLAMD, natural, or an ordering given by the user). Of these ordering choices, the default value in the SUNLinSol_KLU module is the COLAMD ordering.

KLU breaks the factorization into two separate parts. The first is a symbolic factorization and the second is a numeric factorization that returns the factored matrix along with final pivot information. KLU also has a refactor routine that can be called instead of the numeric factorization. This routine will reuse the pivot information. This routine also returns diagnostic information that a user can examine to determine if numerical stability is being lost and a full numerical factorization should be done instead of the refactor.

Since the linear systems that arise within the context of SUNDIALS calculations will typically have identical sparsity patterns, the SUNLinSol_KLU module is constructed to perform the following operations:

- The first time that the "setup" routine is called, it performs the symbolic factorization, followed by an initial numerical factorization.
- On subsequent calls to the "setup" routine, it calls the appropriate KLU "refactor" routine, followed by estimates of the numerical conditioning using the relevant "rcond", and if necessary "condest", routine(s). If these estimates of the condition number are larger than $\varepsilon^{-2/3}$ (where ε is the double-precision unit roundoff), then a new factorization is performed.
- The module includes the routine `SUNKLUReInit`, that can be called by the user to force a full refactorization at the next "setup" call.
- The "solve" call performs pivoting and forward and backward substitution using the stored KLU data structures. We note that in this solve KLU operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

The SUNLinSol_KLU module defines implementations of all "direct" linear solver operations listed in §8.1:

- `SUNLinSolGetType_KLU`
- `SUNLinSolInitialize_KLU` – this sets the `first_factorize` flag to 1, forcing both symbolic and numerical factorizations on the subsequent "setup" call.
- `SUNLinSolSetup_KLU` – this performs either a LU factorization or refactorization of the input matrix.
- `SUNLinSolSolve_KLU` – this calls the appropriate KLU solve routine to utilize the LU factors to solve the linear system.
- `SUNLinSolLastFlag_KLU`

- SUNLinSolSpace_KLU – this only returns information for the storage within the solver *interface*, i.e. storage for the integers last_flag and first_factorize. For additional space requirements, see the KLU documentation.
- SUNLinSolFree_KLU

8.6 The SUNLinSol_LapackBand Module

The SUNLinSol_LapackBand implementation of the SUNLinearSolver class is designed to be used with the corresponding SUNMATRIX_BAND matrix type, and one of the serial or shared-memory N_Vector implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, or NVECTOR_PTHREADS). The

8.6.1 SUNLinSol_LapackBand Usage

The header file to be included when using this module is sunlinsol/sunlinsol_lapackband.h. The installed module library to link to is libsundials_sunlinsollapackband.*lib* where *.lib* is typically .so for shared libraries and .a for static libraries.

The module SUNLinSol_LapackBand provides the following user-callable routine:

SUNLinearSolver **SUNLinSol_LapackBand**(*N_Vector* y, *SUNMatrix* A, *SUNContext* sunctx)
This function creates and allocates memory for a LAPACK band SUNLinearSolver.

Arguments:

- *y* – vector used to determine the linear system size.
- *A* – matrix used to assess compatibility.
- *sunctx* – the *SUNContext* object (see §4.1)

Return value: New SUNLinSol_LapackBand object, or NULL if either A or y are incompatible.

Notes: This routine will perform consistency checks to ensure that it is called with consistent N_Vector 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.

For backwards compatibility, we also provide the following wrapper function:

SUNLinearSolver **SUNLapackBand**(*N_Vector* y, *SUNMatrix* A)
Wrapper function for *SUNLinSol_LapackBand()*, with identical input and output arguments.

8.6.2 SUNLinSol_LapackBand Description

SUNLinSol_LapackBand module defines the *content* field of a `SUNLinearSolver` to be 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.

The SUNLinSol_LapackBand module is a `SUNLinearSolver` 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 §5.2 for details). 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 §10.1.4 for details). We note that since there do not exist 128-bit floating-point factorization and solve routines in LAPACK, this interface cannot be compiled when using `extended` precision for *realtype*. Similarly, since there do not exist 64-bit integer LAPACK routines, the SUNLinSol_LapackBand module also cannot be compiled when using `int64_t` for the *sunindextype*.

This solver is constructed to perform the following operations:

- The "setup" call performs an LU factorization with partial (row) pivoting, $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix with 1's on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX_BAND object A, with pivoting information encoding P stored in the `pivots` array.
- The "solve" call performs pivoting and forward and backward substitution using the stored `pivots` array and the LU factors held in the SUNMATRIX_BAND object.
- A must be allocated to accommodate the increase in upper bandwidth that occurs during factorization. More precisely, if A is a band matrix with upper bandwidth `mu` and lower bandwidth `ml`, then the upper triangular factor U can have upper bandwidth as big as `smu = MIN(N-1,mu+ml)`. The lower triangular factor L has lower bandwidth `ml`.

The SUNLinSol_LapackBand module defines band implementations of all "direct" linear solver operations listed in §8.1:

- `SUNLinSolGetType_LapackBand`
- `SUNLinSolInitialize_LapackBand` – this does nothing, since all consistency checks are performed at solver creation.
- `SUNLinSolSetup_LapackBand` – this calls either `DGBTRF` or `SGBTRF` to perform the LU factorization.
- `SUNLinSolSolve_LapackBand` – this calls either `DGBTRS` or `SGBTRS` to use the LU factors and `pivots` array to perform the solve.
- `SUNLinSolLastFlag_LapackBand`
- `SUNLinSolSpace_LapackBand` – this only returns information for the storage *within* the solver object, i.e. storage for `N`, `last_flag`, and `pivots`.
- `SUNLinSolFree_LapackBand`

8.7 The SUNLinSol_LapackDense Module

The SUNLinSol_LapackDense implementation of the SUNLinearSolver class is designed to be used with the corresponding SUNMATRIX_DENSE matrix type, and one of the serial or shared-memory N_Vector implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, or NVECTOR_PTHREADS).

8.7.1 SUNLinSol_LapackDense Usage

The header file to be included when using this module is sunlinsol/sunlinsol_lapackdense.h. The installed module library to link to is libsundials_sunlinsollapackdense.*lib* where *.lib* is typically .so for shared libraries and .a for static libraries.

The module SUNLinSol_LapackDense provides the following additional user-callable constructor routine:

SUNLinearSolver **SUNLinSol_LapackDense**(*N_Vector* y, *SUNMatrix* A, *SUNContext* sunctx)
This function creates and allocates memory for a LAPACK dense SUNLinearSolver.

Arguments:

- *y* – vector used to determine the linear system size.
- *A* – matrix used to assess compatibility.
- *sunctx* – the *SUNContext* object (see §4.1)

Return value: New SUNLinSol_LapackDense object, or NULL if either A or y are incompatible.

Notes: This routine will perform consistency checks to ensure that it is called with consistent N_Vector 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.

For backwards compatibility, we also provide the following wrapper function:

SUNLinearSolver **SUNLapackDense**(*N_Vector* y, *SUNMatrix* A)
Wrapper function for *SUNLinSol_LapackDense()*, with identical input and output arguments.

8.7.2 SUNLinSol_LapackDense Description

The SUNLinSol_LapackDense module defines the *content* field of a SUNLinearSolver to be 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.

The SUNLinSol_LapackDense module is a SUNLinearSolver 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 §5.2 for details). 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 §10.1.4 for details). We note that since there do not exist 128-bit floating-point factorization and solve routines in LAPACK, this interface cannot be compiled when using extended precision for *realtype*. Similarly, since there do not exist 64-bit integer LAPACK routines, the SUNLinSol_LapackDense module also cannot be compiled when using int64_t for the *sunindextype*.

This solver is constructed to perform the following operations:

- The "setup" call performs an LU factorization with partial (row) pivoting ($\mathcal{O}(N^3)$ cost), $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix with 1's on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX_DENSE object A, with pivoting information encoding P stored in the pivots array.
- The "solve" call performs pivoting and forward and backward substitution using the stored pivots array and the LU factors held in the SUNMATRIX_DENSE object ($\mathcal{O}(N^2)$ cost).

The SUNLinSol_LapackDense module defines dense implementations of all "direct" linear solver operations listed in §8.1:

- SUNLinSolGetType_LapackDense
- SUNLinSolInitialize_LapackDense – this does nothing, since all consistency checks are performed at solver creation.
- SUNLinSolSetup_LapackDense – this calls either DGETRF or SGETRF to perform the LU factorization.
- SUNLinSolSolve_LapackDense – this calls either DGETRS or SGETRS to use the LU factors and pivots array to perform the solve.
- SUNLinSolLastFlag_LapackDense
- SUNLinSolSpace_LapackDense – this only returns information for the storage *within* the solver object, i.e. storage for N, last_flag, and pivots.
- SUNLinSolFree_LapackDense

8.8 The SUNLinSol_MagmaDense Module

The SUNLinearSolver_MagmaDense implementation of the SUNLinearSolver class is designed to be used with the SUNMATRIX_MAGMADENSE matrix, and a GPU-enabled vector. 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.

Warning: The SUNLinearSolver_MagmaDense module is experimental and subject to change.

8.8.1 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 [32]. The batched LU methods are leveraged when solving block diagonal linear systems of the form

$$\begin{bmatrix} \mathbf{A_0} & 0 & \cdots & 0 \\ 0 & \mathbf{A_1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A_{n-1}} \end{bmatrix} x_j = b_j.$$

8.8.2 SUNLinearSolver_MagmaDense Functions

The SUNLinearSolver_MagmaDense module defines implementations of all "direct" linear solver operations listed in §8.1:

- `SUNLinSolGetType_MagmaDense`
- `SUNLinSolInitialize_MagmaDense`
- `SUNLinSolSetup_MagmaDense`
- `SUNLinSolSolve_MagmaDense`
- `SUNLinSolLastFlag_MagmaDense`
- `SUNLinSolFree_MagmaDense`

In addition, the module provides the following user-callable routines:

SUNLinearSolver **SUNLinSol_MagmaDense**(*N_Vector* y, *SUNMatrix* A, *SUNContext* sunctx)
This constructor function creates and allocates memory for a `SUNLinearSolver` object.

Arguments:

- *y* – a vector for checking compatibility with the solver.
- *A* – a SUNMATRIX_MAGMADENSE matrix for checking compatibility with the solver.
- *sunctx* – the *SUNContext* object (see §4.1)

Return value: If successful, a `SUNLinearSolver` object. If either *A* or *y* are incompatible then this routine will return `NULL`. This routine analyzes the input matrix and vector to determine the linear system size and to assess compatibility with the solver.

int **SUNLinSol_MagmaDense_SetAsync**(*SUNLinearSolver* LS, *booleantype* onoff)
This function can be used to toggle the linear solver between asynchronous and synchronous modes. In asynchronous mode (default), 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* – a SUNLinSol_MagmaDense object
- *onoff* – 0 for synchronous mode or 1 for asynchronous mode (default 1)

Return value:

- `SUNLS_SUCCESS` if successful
- `SUNLS_MEM_NULL` if *LS* is `NULL`

8.8.3 SUNLinearSolver_MagmaDense Content

The SUNLinearSolver_MagmaDense module defines the object *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;
};
```

8.9 The SUNLinSol_OneMklDense Module

The SUNLinearSolver_OneMklDense implementation of the `SUNLinearSolver` class interfaces to the direct linear solvers from the Intel oneAPI Math Kernel Library (oneMKL) for solving dense systems or 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.

Warning: The SUNLinearSolver_OneMklDense class is experimental and subject to change.

8.9.1 SUNLinearSolver_OneMklDense Functions

The SUNLinearSolver_OneMklDense class defines implementations of all "direct" linear solver operations listed in §8.1:

- `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:

SUNLinearSolver **SUNLinSol_OneMklDense**(*N_Vector* y, *SUNMatrix* A, *SUNContext* sunctx)
This constructor function creates and allocates memory for a `SUNLinearSolver` object.

Arguments:

- *y* – a vector for checking compatibility with the solver.

- *A* – a SUNMatrix_OneMklDense matrix for checking compatibility with the solver.
- *sunctx* – the *SUNContext* object (see §4.1)

Return value: If successful, a SUNLinearSolver object. If either *A* or *y* are incompatible then this routine will return NULL. This routine analyzes the input matrix and vector to determine the linear system size and to assess compatibility with the solver.

8.9.2 SUNLinearSolver_OneMklDense Usage Notes

Warning: 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 the matrix is also used for the linear solver.

8.10 The SUNLinSol_PCG Module

The SUNLinSol_PCG implementation of the SUNLinearSolver class performs the PCG (Preconditioned Conjugate Gradient [19]) method; this is an iterative linear solver that is designed to be compatible with any N_Vector 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.

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 N_Vector, supplied by the user.

In this notation, PCG applies the underlying CG algorithm to the equivalent transformed system

$$\tilde{A}\tilde{x} = \tilde{b} \tag{8.3}$$

where

$$\begin{aligned} \tilde{A} &= SP^{-1}AP^{-1}S, \\ \tilde{b} &= SP^{-1}b, \\ \tilde{x} &= S^{-1}Px. \end{aligned} \tag{8.4}$$

The scaling matrix must be chosen so that the vectors $SP^{-1}b$ and $S^{-1}Px$ have dimensionless components.

The stopping test for the PCG iterations is on the L2 norm of the scaled preconditioned residual:

$$\begin{aligned} & \|\tilde{b} - \tilde{A}\tilde{x}\|_2 < \delta \\ \Leftrightarrow & \\ & \|SP^{-1}b - SP^{-1}Ax\|_2 < \delta \\ \Leftrightarrow & \\ & \|P^{-1}b - P^{-1}Ax\|_S < \delta \end{aligned}$$

where $\|v\|_S = \sqrt{v^T S^T S v}$, with an input tolerance δ.

8.10.1 SUNLinSol_PCG Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_pcg.h`. The SUNLinSol_PCG module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolpcg` module library.

The module SUNLinSol_PCG provides the following user-callable routines:

SUNLinearSolver **SUNLinSol_PCG**(*N_Vector* y, int pretype, int maxl, *SUNContext* sunctx)

This constructor function creates and allocates memory for a PCG `SUNLinearSolver`.

Arguments:

- *y* – a template vector.
- *pretype* – a flag indicating the type of preconditioning to use:
  - `SUN_PREC_NONE`
  - `SUN_PREC_LEFT`
  - `SUN_PREC_RIGHT`
  - `SUN_PREC_BOTH`
- *maxl* – the maximum number of linear iterations to allow.
- *sunctx* – the *SUNContext* object (see §4.1)

Return value: If successful, 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 `N_Vector` implementation (i.e. that it supplies the requisite vector operations).

A `maxl` argument that is ≤ 0 will result in the default value (5).

Since the PCG algorithm is designed to only support symmetric preconditioning, then any of the `pretype` inputs `SUN_PREC_LEFT`, `SUN_PREC_RIGHT`, or `SUN_PREC_BOTH` will result in use of the symmetric preconditioner; any other integer input will result in the default (no preconditioning). Although some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL), PCG should *only* be used with these packages when the linear systems are known to be *symmetric*. Since the scaling of matrix rows and columns must be identical in a symmetric matrix, symmetric preconditioning should work appropriately even for packages designed with one-sided preconditioning in mind.

int **SUNLinSol_PCGSetPrecType**(*SUNLinearSolver* S, int pretype)

This function updates the flag indicating use of preconditioning.

Arguments:

- *S* – SUNLinSol_PCG object to update.

- *pretype* – a flag indicating the type of preconditioning to use:
  - SUN_PREC_NONE
  - SUN_PREC_LEFT
  - SUN_PREC_RIGHT
  - SUN_PREC_BOTH

Return value:

- SUNLS_SUCCESS – successful update.
- SUNLS_ILL_INPUT – illegal pretype
- SUNLS_MEM_NULL – S is NULL

Notes: As above, any one of the input values, SUN_PREC_LEFT, SUN_PREC_RIGHT, or SUN_PREC_BOTH will enable preconditioning; SUN_PREC_NONE disables preconditioning.

int **SUNLinSol_PCGSetMaxl**(*SUNLinearSolver* S, int maxl)
This function updates the number of linear solver iterations to allow.

Arguments:

- *S* – SUNLinSol_PCG object to update.
- *maxl* – maximum number of linear iterations to allow. Any non-positive input will result in the default value (5).

Return value:

- SUNLS_SUCCESS – successful update.
- SUNLS_MEM_NULL – S is NULL

int **SUNLinSolSetInfoFile_PCG**(*SUNLinearSolver* LS, FILE *info_file)
The function *SUNLinSolSetInfoFile_PCG()* sets the output file where all informative (non-error) messages should be directed.

Arguments:

- *LS* – a SUNLinSol object
- ***info_file* – pointer to output file (stdout by default);** a NULL input will disable output

Return value:

- *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 §10.1.2 for more information.

int **SUNLinSolSetPrintLevel_PCG**(*SUNLinearSolver* LS, int print_level)
The function *SUNLinSolSetPrintLevel_PCG()* specifies the level of verbosity of the output.

Arguments:

- *LS* – a SUNLinSol object
- *print_level* – 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:

- *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 if 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 §10.1.2 for more information.

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

SUNLinearSolver **SUNPCG**(*N_Vector* y, int pretype, int maxl)
Wrapper function for *SUNLinSol_PCG()*

int **SUNPCGSetPrecType**(*SUNLinearSolver* S, int pretype)
Wrapper function for *SUNLinSol_PCGSetPrecType()*

int **SUNPCGSetMaxl**(*SUNLinearSolver* S, int maxl)
Wrapper function for *SUNLinSol_PCGSetMaxl()*

8.10.2 SUNLinSol_PCG Description

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

```
struct _SUNLinearSolverContent_PCG {
  int maxl;
  int pretype;
  booleantype zeroguess;
  int numiters;
  realtype resnorm;
  int last_flag;
  SUNATimesFn ATimes;
  void* ATData;
  SUNPSetupFn Psetup;
  SUNPSolveFn 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 `N_Vector` which holds the preconditioned linear system residual,
- `p, z, Ap` - `N_Vector` 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

This solver is constructed to perform the following operations:

- During construction all `N_Vector` solver data is allocated, with vectors cloned from a template `N_Vector` that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLinSol_PCG to supply the `ATimes`, `PSetup`, and `Psolve` function pointers and `s` scaling vector.
- In the "initialize" call, the solver parameters are checked for validity.
- In the "setup" call, any non-`NULL` `PSetup` function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call the PCG iteration is performed. This will include scaling and preconditioning if those options have been supplied.

The SUNLinSol_PCG module defines implementations of all "iterative" linear solver operations listed in §8.1:

- `SUNLinSolGetType_PCG`
- `SUNLinSolInitialize_PCG`
- `SUNLinSolSetATimes_PCG`
- `SUNLinSolSetPreconditioner_PCG`
- `SUNLinSolSetScalingVectors_PCG` – since PCG only supports symmetric scaling, the second `N_Vector` 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

8.11 The SUNLinSol_SPBCGS Module

The SUNLinSol_SPBCGS implementation of the SUNLinearSolver class performs a Scaled, Preconditioned, Bi-Conjugate Gradient, Stabilized [33] method; this is an iterative linear solver that is designed to be compatible with any N_Vector 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.

8.11.1 SUNLinSol_SPBCGS Usage

The header file to be included when using this module is sunlinsol/sunlinsol_spbcgs.h. The SUNLinSol_SPBCGS module is accessible from all SUNDIALS solvers *without* linking to the libsundials_sunlinsolspbcgs module library.

The module SUNLinSol_SPBCGS provides the following user-callable routines:

SUNLinearSolver **SUNLinSol_SPBCGS**(*N_Vector* y, int pretype, int maxl, *SUNContext* sunctx)

This constructor function creates and allocates memory for a SPBCGS SUNLinearSolver.

Arguments:

- *y* – a template vector.
- *pretype* – a flag indicating the type of preconditioning to use:
  - SUN_PREC_NONE
  - SUN_PREC_LEFT
  - SUN_PREC_RIGHT
  - SUN_PREC_BOTH
- *maxl* – the maximum number of linear iterations to allow.
- *sunctx* – the *SUNContext* object (see §4.1)

Return value: If successful, 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 N_Vector implementation (i.e. that it supplies the requisite vector operations).

A maxl argument that is ≤ 0 will result in the default value (5).

Some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL). While it is possible to configure a SUNLinSol_SPBCGS object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.

Note: With SUN_PREC_RIGHT or SUN_PREC_BOTH the initial guess must be zero (use *SUNLinSolSetZeroGuess()* to indicate the initial guess is zero).

int **SUNLinSol_SPBCGSSetPrecType**(*SUNLinearSolver* S, int pretype)
This function updates the flag indicating use of preconditioning.

Arguments:

- *S* – SUNLinSol_SPBCGS object to update.
- *pretype* – a flag indicating the type of preconditioning to use:
  - SUN_PREC_NONE
  - SUN_PREC_LEFT
  - SUN_PREC_RIGHT
  - SUN_PREC_BOTH

Return value:

- SUNLS_SUCCESS – successful update.
- SUNLS_ILL_INPUT – illegal pretype
- SUNLS_MEM_NULL – S is NULL

int **SUNLinSol_SPBCGSSetMaxl**(*SUNLinearSolver* S, int maxl)
This function updates the number of linear solver iterations to allow.

Arguments:

- *S* – SUNLinSol_SPBCGS object to update.
- *maxl* – maximum number of linear iterations to allow. Any non-positive input will result in the default value (5).

Return value:

- SUNLS_SUCCESS – successful update.
- SUNLS_MEM_NULL – S is NULL

int **SUNLinSolSetInfoFile_SPBCGS**(*SUNLinearSolver* LS, FILE *info_file)
The function *SUNLinSolSetInfoFile_SPBCGS()* sets the output file where all informative (non-error) messages should be directed.

Arguments:

- *LS* – a SUNLinSol object
- ***info_file* – pointer to output file (stdout by default);** a NULL input will disable output

Return value:

- *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 §10.1.2 for more information.

int **SUNLinSolSetPrintLevel_SPBCGS**(*SUNLinearSolver* LS, int print_level)
The function *SUNLinSolSetPrintLevel_SPBCGS()* specifies the level of verbosity of the output.

Arguments:

- *LS* – a SUNLinSol object
- *print_level* – 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:

- *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 if 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 §10.1.2 for more information.

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

SUNLinearSolver **SUNSPBCGS**(*N_Vector* y, int pretype, int maxl)
Wrapper function for *SUNLinSol_SPBCGS()*

int **SUNSPBCGSSetPrecType**(*SUNLinearSolver* S, int pretype)
Wrapper function for *SUNLinSol_SPBCGSSetPrecType()*

int **SUNSPBCGSSetMaxl**(*SUNLinearSolver* S, int maxl)
Wrapper function for *SUNLinSol_SPBCGSSetMaxl()*

8.11.2 SUNLinSol_SPBCGS Description

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

```
struct _SUNLinearSolverContent_SPBCGS {
  int maxl;
  int pretype;
  booleantype zeroguess;
  int numiters;
  realtype resnorm;
  int last_flag;
  SUNATimesFn ATimes;
  void* ATData;
  SUNPSetupFn Psetup;
  SUNPSolveFn Psolve;
  void* PData;
  N_Vector s1;
```

(continues on next page)

(continued from previous page)

```
  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 `N_Vector` which holds the current scaled, preconditioned linear system residual,
- `r_star` - a `N_Vector` which holds the initial scaled, preconditioned linear system residual,
- `p, q, u, Ap, vtemp` - `N_Vector` 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

This solver is constructed to perform the following operations:

- During construction all `N_Vector` solver data is allocated, with vectors cloned from a template `N_Vector` that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLinSol_SPBCGS to supply the `ATimes`, `PSetup`, and `Psolve` function pointers and `s1` and `s2` scaling vectors.
- In the "initialize" call, the solver parameters are checked for validity.
- In the "setup" call, any non-`NULL` `PSetup` function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call the SPBCGS iteration is performed. This will include scaling and preconditioning if those options have been supplied.

The SUNLinSol_SPBCGS module defines implementations of all "iterative" linear solver operations listed in §8.1:

- 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

8.12 The SUNLinSol_SPFGMR Module

The SUNLinSol_SPFGMR implementation of the SUNLinearSolver class performs a Scaled, Preconditioned, Flexible, Generalized Minimum Residual [29] method; this is an iterative linear solver that is designed to be compatible with any N_Vector 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 other Krylov iterative linear solvers supplied with SUNDIALS, FGMRES is specifically designed to work with a changing preconditioner (e.g. from an iterative method).

8.12.1 SUNLinSol_SPFGMR Usage

The header file to be included when using this module is sunlinsol/sunlinsol_spfgmr.h. The SUNLinSol_SPFGMR module is accessible from all SUNDIALS solvers *without* linking to the libsundials_sunlinsolspfgmr module library.

The module SUNLinSol_SPFGMR provides the following user-callable routines:

SUNLinearSolver **SUNLinSol_SPFGMR**(*N_Vector* y, int pretype, int maxl, *SUNContext* sunctx)
This constructor function creates and allocates memory for a SPFGMR SUNLinearSolver.

Arguments:

- *y* – a template vector.
- *pretype* – a flag indicating the type of preconditioning to use:
  - SUN_PREC_NONE
  - SUN_PREC_LEFT
  - SUN_PREC_RIGHT

- SUN_PREC_BOTH

- *maxl* – the number of Krylov basis vectors to use.
- *sunctx* – the *SUNContext* object (see §4.1)

Return value: If successful, 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 N_Vector implementation (i.e. that it supplies the requisite vector operations).

A maxl argument that is ≤ 0 will result in the default value (5).

Since the FGMRES algorithm is designed to only support right preconditioning, then any of the pretype inputs SUN_PREC_LEFT, SUN_PREC_RIGHT, or SUN_PREC_BOTH will result in use of SUN_PREC_RIGHT; any other integer input will result in the default (no preconditioning). We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS). While it is possible to use a right-preconditioned SUNLinSol_SPFGMR object for these packages, this use mode is not supported and may result in inferior performance.

int **SUNLinSol_SPFGMRSetPrecType**(*SUNLinearSolver* S, int pretype)

This function updates the flag indicating use of preconditioning.

Arguments:

- *S* – SUNLinSol_SPFGMR object to update.
- *pretype* – a flag indicating the type of preconditioning to use:
  - SUN_PREC_NONE
  - SUN_PREC_LEFT
  - SUN_PREC_RIGHT
  - SUN_PREC_BOTH

Return value:

- SUNLS_SUCCESS – successful update.
- SUNLS_ILL_INPUT – illegal pretype
- SUNLS_MEM_NULL – S is NULL

Notes: Since the FGMRES algorithm is designed to only support right preconditioning, then any of the pretype inputs SUN_PREC_LEFT, SUN_PREC_RIGHT, or SUN_PREC_BOTH will result in use of SUN_PREC_RIGHT; any other integer input will result in the default (no preconditioning).

int **SUNLinSol_SPFGMRSetGSType**(*SUNLinearSolver* S, int gstype)

This function sets the type of Gram-Schmidt orthogonalization to use.

Arguments:

- *S* – SUNLinSol_SPFGMR object to update.
- *gstype* – a flag indicating the type of orthogonalization to use:
  - SUN_MODIFIED_GS
  - SUN_CLASSICAL_GS

Return value:

- SUNLS_SUCCESS – successful update.
- SUNLS_ILL_INPUT – illegal gstype

- SUNLS_MEM_NULL – S is NULL

int **SUNLinSol_SPFGMRSetMaxRestarts**(*SUNLinearSolver* S, int maxrs)

This function sets the number of FGMRES restarts to allow.

Arguments:

- *S* – SUNLinSol_SPFGMR object to update.
- *maxrs* – maximum number of restarts to allow. A negative input will result in the default of 0.

Return value:

- SUNLS_SUCCESS – successful update.
- SUNLS_MEM_NULL – S is NULL

int **SUNLinSolSetInfoFile_SPFGMR**(*SUNLinearSolver* LS, FILE *info_file)

The function *SUNLinSolSetInfoFile_SPFGMR()* sets the output file where all informative (non-error) messages should be directed.

Arguments:

- *LS* – a SUNLinSol object
- ***info_file* – pointer to output file (stdout by default);** a NULL input will disable output

Return value:

- *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 §10.1.2 for more information.

int **SUNLinSolSetPrintLevel_SPFGMR**(*SUNLinearSolver* LS, int print_level)

The function *SUNLinSolSetPrintLevel_SPFGMR()* specifies the level of verbosity of the output.

Arguments:

- *LS* – a SUNLinSol object
- *print_level* – 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:

- *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 if 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 §10.1.2 for more information.

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

SUNLinearSolver **SUNSPFGMR**(*N_Vector* y, int pretype, int maxl)
Wrapper function for *SUNLinSol_SPFGMR()*

int **SUNSPFGMRSetPrecType**(*SUNLinearSolver* S, int pretype)
Wrapper function for *SUNLinSol_SPFGMRSetPrecType()*

int **SUNSPFGMRSetGSType**(*SUNLinearSolver* S, int gstype)
Wrapper function for *SUNLinSol_SPFGMRSetGSType()*

int **SUNSPFGMRSetMaxRestarts**(*SUNLinearSolver* S, int maxrs)
Wrapper function for *SUNLinSol_SPFGMRSetMaxRestarts()*

8.12.2 SUNLinSol_SPFGMR Description

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

```
struct _SUNLinearSolverContent_SPFGMR {
  int maxl;
  int pretype;
  int gstype;
  int max_restarts;
  booleantype zeroguess;
  int numiters;
  realtype resnorm;
  int last_flag;
  SUNATimesFn ATimes;
  void* ATData;
  SUNPSetupFn Psetup;
  SUNPSolveFn 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:

- `maxl` - number of FGMRES basis vectors to use (default is 5),
- `pretype` - flag for use of preconditioning (default is none),
- `gstype` - flag for type of Gram-Schmidt orthogonalization (default is modified Gram-Schmidt),
- `max_restarts` - number of FGMRES restarts to allow (default is 0),
- `numiters` - number of iterations from the most-recent solve,

- `resnorm` - final linear residual norm from the most-recent solve,
- `last_flag` - last error return flag from an internal function,
- `ATimes` - function pointer to perform Av product,
- `ATData` - pointer to structure for `ATimes`,
- `Psetup` - function pointer to preconditioner setup routine,
- `Psolve` - function pointer to preconditioner solve routine,
- `PData` - pointer to structure for `Psetup` and `Psolve`,
- `s1, s2` - vector pointers for supplied scaling matrices (default is `NULL`),
- `V` - the array of Krylov basis vectors $v_1, \ldots, v_{\text{maxl}+1}$, stored in `V[0], ..., V[maxl]`. Each v_i is a vector of type `N_Vector`,
- `Z` - the array of preconditioned Krylov basis vectors $z_1, \ldots, z_{\text{maxl}+1}$, stored in `Z[0], ..., Z[maxl]`. Each z_i is a vector of type `N_Vector`,
- `Hes` - the $(\text{maxl} + 1) \times$ maxl Hessenberg matrix. It is stored row-wise so that the (i,j)th element is given by `Hes[i][j]`,
- `givens` - a length $2\,$maxl array which represents the Givens rotation matrices that arise in the FGMRES algorithm. These matrices are $F_0, F_1, \ldots, F_j$, where

$$F_i = \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_i & -s_i & & & \\ & & & s_i & c_i & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix},$$

  are represented in the `givens` vector as `givens[0]` $= c_0$, `givens[1]` $= s_0$, `givens[2]` $= c_1$, `givens[3]` $= s_1$, ..., `givens[2j]` $= c_j$, `givens[2j+1]` $= s_j$,
- `xcor` - a vector which holds the scaled, preconditioned correction to the initial guess,
- `yg` - a length $(\text{maxl} + 1)$ array of `realtype` 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

This solver is constructed to perform the following operations:

- During construction, the `xcor` and `vtemp` arrays are cloned from a template `N_Vector` that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLinSol_SPFGMR to supply the `ATimes`, `PSetup`, and `Psolve` function pointers and `s1` and `s2` scaling vectors.
- In the "initialize" call, the remaining solver data is allocated (`V`, `Hes`, `givens`, and `yg`)
- In the "setup" call, any non-`NULL` `PSetup` function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).

- In the "solve" call, the FGMRES iteration is performed. This will include scaling, preconditioning, and restarts if those options have been supplied.

The SUNLinSol_SPFGMR module defines implementations of all "iterative" linear solver operations listed in §8.1:

- 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

8.13 The SUNLinSol_SPGMR Module

The SUNLinSol_SPGMR implementation of the SUNLinearSolver class performs a Scaled, Preconditioned, Generalized Minimum Residual [30] method; this is an iterative linear solver that is designed to be compatible with any N_Vector 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()*).

8.13.1 SUNLinSol_SPGMR Usage

The header file to be included when using this module is sunlinsol/sunlinsol_spgmr.h. The SUNinSol_SPGMR module is accessible from all SUNDIALS solvers *without* linking to the libsundials_sunlinsolspgmr module library.

The module SUNLinSol_SPGMR provides the following user-callable routines:

SUNLinearSolver **SUNLinSol_SPGMR**(*N_Vector* y, int pretype, int maxl, *SUNContext* sunctx)
This constructor function creates and allocates memory for a SPGMR SUNLinearSolver.

Arguments:

- *y* – a template vector.
- *pretype* – a flag indicating the type of preconditioning to use:
  - SUN_PREC_NONE
  - SUN_PREC_LEFT
  - SUN_PREC_RIGHT

- SUN_PREC_BOTH

- *maxl* – the number of Krylov basis vectors to use.

Return value: If successful, 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 N_Vector implementation (i.e. that it supplies the requisite vector operations).

A maxl argument that is ≤ 0 will result in the default value (5).

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.

int **SUNLinSol_SPGMRSetPrecType**(*SUNLinearSolver* S, int pretype)

This function updates the flag indicating use of preconditioning.

Arguments:

- *S* – SUNLinSol_SPGMR object to update.
- *pretype* – a flag indicating the type of preconditioning to use:
  - SUN_PREC_NONE
  - SUN_PREC_LEFT
  - SUN_PREC_RIGHT
  - SUN_PREC_BOTH

Return value:

- SUNLS_SUCCESS – successful update.
- SUNLS_ILL_INPUT – illegal pretype
- SUNLS_MEM_NULL – S is NULL

int **SUNLinSol_SPGMRSetGSType**(*SUNLinearSolver* S, int gstype)

This function sets the type of Gram-Schmidt orthogonalization to use.

Arguments:

- *S* – SUNLinSol_SPGMR object to update.
- *gstype* – a flag indicating the type of orthogonalization to use:
  - SUN_MODIFIED_GS
  - SUN_CLASSICAL_GS

Return value:

- SUNLS_SUCCESS – successful update.
- SUNLS_ILL_INPUT – illegal gstype
- SUNLS_MEM_NULL – S is NULL

int **SUNLinSol_SPGMRSetMaxRestarts**(*SUNLinearSolver* S, int maxrs)

This function sets the number of GMRES restarts to allow.

Arguments:

- *S* – SUNLinSol_SPGMR object to update.

- *maxrs* – maximum number of restarts to allow. A negative input will result in the default of 0.

Return value:

- SUNLS_SUCCESS – successful update.
- SUNLS_MEM_NULL – S is NULL

int **SUNLinSolSetInfoFile_SPGMR**(*SUNLinearSolver* LS, FILE *info_file)
The function *SUNLinSolSetInfoFile_SPGMR()* sets the output file where all informative (non-error) messages should be directed.

Arguments:

- *LS* – a SUNLinSol object
- *info_file* – pointer to output file (stdout by default); a NULL input will disable output

Return value:

- *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 §10.1.2 for more information.

int **SUNLinSolSetPrintLevel_SPGMR**(*SUNLinearSolver* LS, int print_level)
The function *SUNLinSolSetPrintLevel_SPGMR()* specifies the level of verbosity of the output.

Arguments:

- *LS* – a SUNLinSol object
- *print_level* – 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:

- *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 if 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 §10.1.2 for more information.

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

SUNLinearSolver **SUNSPGMR**(*N_Vector* y, int pretype, int maxl)
Wrapper function for *SUNLinSol_SPGMR()*

int **SUNSPGMRSetPrecType**(*SUNLinearSolver* S, int pretype)
Wrapper function for *SUNLinSol_SPGMRSetPrecType()*

int **SUNSPGMRSetGSType**(*SUNLinearSolver* S, int gstype)
Wrapper function for *SUNLinSol_SPGMRSetGSType()*

int **SUNSPGMRSetMaxRestarts**(*SUNLinearSolver* S, int maxrs)
Wrapper function for *SUNLinSol_SPGMRSetMaxRestarts()*

8.13.2 SUNLinSol_SPGMR Description

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

```
struct _SUNLinearSolverContent_SPGMR {
  int maxl;
  int pretype;
  int gstype;
  int max_restarts;
  booleantype zeroguess;
  int numiters;
  realtype resnorm;
  int last_flag;
  SUNATimesFn ATimes;
  void* ATData;
  SUNPSetupFn Psetup;
  SUNPSolveFn 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:

- `maxl` - number of GMRES basis vectors to use (default is 5),
- `pretype` - flag for type of preconditioning to employ (default is none),
- `gstype` - flag for type of Gram-Schmidt orthogonalization (default is modified Gram-Schmidt),
- `max_restarts` - number of GMRES restarts to allow (default is 0),
- `numiters` - number of iterations from the most-recent solve,
- `resnorm` - final linear residual norm from the most-recent solve,
- `last_flag` - last error return flag from an internal function,
- `ATimes` - function pointer to perform Av product,
- `ATData` - pointer to structure for `ATimes`,
- `Psetup` - function pointer to preconditioner setup routine,
- `Psolve` - function pointer to preconditioner solve routine,

- `PData` - pointer to structure for `Psetup` and `Psolve`,
- `s1, s2` - vector pointers for supplied scaling matrices (default is `NULL`),
- `V` - the array of Krylov basis vectors $v_1, \ldots, v_{\text{maxl}+1}$, stored in `V[0], ... V[maxl]`. Each v_i is a vector of type `N_Vector`,
- `Hes` - the $(\text{maxl}+1) \times \text{maxl}$ Hessenberg matrix. It is stored row-wise so that the (i,j)th element is given by `Hes[i][j]`,
- `givens` - a length 2 maxl array which represents the Givens rotation matrices that arise in the GMRES algorithm. These matrices are $F_0, F_1, \ldots, F_j$, where

$$F_i = \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_i & -s_i & & & \\ & & & s_i & c_i & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix},$$

  are represented in the `givens` vector as `givens[0]` $= c_0$, `givens[1]` $= s_0$, `givens[2]` $= c_1$, `givens[3]` $= s_1$, ..., `givens[2j]` $= c_j$, `givens[2j+1]` $= s_j$,
- `xcor` - a vector which holds the scaled, preconditioned correction to the initial guess,
- `yg` - a length $(\text{maxl}+1)$ array of `realtype` 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

This solver is constructed to perform the following operations:

- During construction, the `xcor` and `vtemp` arrays are cloned from a template `N_Vector` that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLinSol_SPGMR to supply the `ATimes`, `PSetup`, and `Psolve` function pointers and `s1` and `s2` scaling vectors.
- In the "initialize" call, the remaining solver data is allocated (`V`, `Hes`, `givens`, and `yg`)
- In the "setup" call, any non-`NULL` `PSetup` function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call, the GMRES iteration is performed. This will include scaling, preconditioning, and restarts if those options have been supplied.

The SUNLinSol_SPGMR module defines implementations of all "iterative" linear solver operations listed in §8.1:

- `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

8.14 The SUNLinSol_SPTFQMR Module

The SUNLinSol_SPTFQMR implementation of the SUNLinearSolver class performs a Scaled, Preconditioned, Transpose-Free Quasi-Minimum Residual [16] method; this is an iterative linear solver that is designed to be compatible with any N_Vector 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.

8.14.1 SUNLinSol_SPTFQMR Usage

The header file to be included when using this module is sunlinsol/sunlinsol_sptfqmr.h. The SUNLinSol_SPTFQMR module is accessible from all SUNDIALS solvers *without* linking to the libsundials_sunlinsolsptfqmr module library.

The module SUNLinSol_SPTFQMR provides the following user-callable routines:

SUNLinearSolver **SUNLinSol_SPTFQMR**(*N_Vector* y, int pretype, int maxl, *SUNContext* sunctx)
This constructor function creates and allocates memory for a SPTFQMR SUNLinearSolver.

Arguments:

- *y* – a template vector.
- *pretype* – a flag indicating the type of preconditioning to use:
  - SUN_PREC_NONE
  - SUN_PREC_LEFT
  - SUN_PREC_RIGHT
  - SUN_PREC_BOTH
- *maxl* – the number of Krylov basis vectors to use.
- *sunctx* – the *SUNContext* object (see §4.1)

Return value: If successful, 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 `N_Vector` implementation (i.e. that it supplies the requisite vector operations).

A `maxl` argument that is ≤ 0 will result in the default value (5).

Some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL). While it is possible to configure a SUNLinSol_SPTFQMR object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.

Note: With `SUN_PREC_RIGHT` or `SUN_PREC_BOTH` the initial guess must be zero (use *SUNLinSolSetZeroGuess()* to indicate the initial guess is zero).

---

int **SUNLinSol_SPTFQMRSetPrecType**(*SUNLinearSolver* S, int pretype)

This function updates the flag indicating use of preconditioning.

Arguments:

- *S* – SUNLinSol_SPGMR object to update.
- *pretype* – a flag indicating the type of preconditioning to use:
  - `SUN_PREC_NONE`
  - `SUN_PREC_LEFT`
  - `SUN_PREC_RIGHT`
  - `SUN_PREC_BOTH`

Return value:

- `SUNLS_SUCCESS` – successful update.
- `SUNLS_ILL_INPUT` – illegal `pretype`
- `SUNLS_MEM_NULL` – `S` is `NULL`

int **SUNLinSol_SPTFQMRSetMaxl**(*SUNLinearSolver* S, int maxl)

This function updates the number of linear solver iterations to allow.

Arguments:

- *S* – SUNLinSol_SPTFQMR object to update.
- *maxl* – maximum number of linear iterations to allow. Any non-positive input will result in the default value (5).

Return value:

- `SUNLS_SUCCESS` – successful update.
- `SUNLS_MEM_NULL` – `S` is `NULL`

int **SUNLinSolSetInfoFile_SPTFQMR**(*SUNLinearSolver* LS, FILE *info_file)

The function *SUNLinSolSetInfoFile_SPTFQMR()* sets the output file where all informative (non-error) messages should be directed.

Arguments:

- *LS* – a SUNLinSol object
- ***info_file* – pointer to output file (`stdout` by default);** a `NULL` input will disable output

Return value:

- *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 §10.1.2 for more information.

int **SUNLinSolSetPrintLevel_SPTFQMR**(*SUNLinearSolver* LS, int print_level)
The function *SUNLinSolSetPrintLevel_SPTFQMR()* specifies the level of verbosity of the output.

Arguments:

- *LS* – a SUNLinSol object
- *print_level* – 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:

- *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 if 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 §10.1.2 for more information.

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

SUNLinearSolver **SUNSPTFQMR**(*N_Vector* y, int pretype, int maxl)
Wrapper function for *SUNLinSol_SPTFQMR()*

int **SUNSPTFQMRSetPrecType**(*SUNLinearSolver* S, int pretype)
Wrapper function for *SUNLinSol_SPTFQMRSetPrecType()*

int **SUNSPTFQMRSetMaxl**(*SUNLinearSolver* S, int maxl)
Wrapper function for *SUNLinSol_SPTFQMRSetMaxl()*

8.14.2 SUNLinSol_SPTFQMR Description

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

```
struct _SUNLinearSolverContent_SPTFQMR {
  int maxl;
  int pretype;
  booleantype zeroguess;
  int numiters;
  realtype resnorm;
```

(continues on next page)

(continued from previous page)

```
  int last_flag;
  SUNATimesFn ATimes;
  void* ATData;
  SUNPSetupFn Psetup;
  SUNPSolveFn 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 `N_Vector` which holds the initial scaled, preconditioned linear system residual,
- `q, d, v, p, u` - `N_Vector` used for workspace by the SPTFQMR algorithm,
- `r` - array of two `N_Vector` 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

This solver is constructed to perform the following operations:

- During construction all `N_Vector` solver data is allocated, with vectors cloned from a template `N_Vector` that is input, and default solver parameters are set.

- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLinSol_SPTFQMR to supply the `ATimes`, `PSetup`, and `Psolve` function pointers and `s1` and `s2` scaling vectors.
- In the "initialize" call, the solver parameters are checked for validity.
- In the "setup" call, any non-`NULL` `PSetup` function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call the TFQMR iteration is performed. This will include scaling and preconditioning if those options have been supplied.

The SUNLinSol_SPTFQMR module defines implementations of all "iterative" linear solver operations listed in §8.1:

- `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`

8.15 The SUNLinSol_SuperLUDIST Module

The SUNLinsol_SuperLUDIST implementation of the `SUNLinearSolver` class interfaces with the SuperLU_DIST library. This is designed to be used with the SUNMatrix_SLUNRloc *SUNMatrix*, and one of the serial, threaded or parallel N_Vector implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, NVECTOR_PTHREADS, NVECTOR_PARALLEL, NVECTOR_PARHYP).

8.15.1 SUNLinSol_SuperLUDIST Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_superludist.h`. The installed module library to link to is `libsundials_sunlinsolsuperludist.`*lib* where *.lib* is typically `.so` for shared libraries and `.a` for static libraries.

The module SUNLinSol_SuperLUDIST provides the following user-callable routines:

Warning: 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 unprefixed 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.

SUNLinearSolver **SUNLinSol_SuperLUDIST**(*N_Vector* y, SuperMatrix *A, gridinfo_t *grid, xLUstruct_t *lu, xScalePermstruct_t *scaleperm, xSOLVEstruct_t *solve, SuperLUStat_t *stat, superlu_dist_options_t *options, *SUNContext* sunctx)

This constructor function creates and allocates memory for a SUNLinSol_SuperLUDIST object.

Arguments:

- *y* – a template vector.
- *A* – a template matrix
- *grid*, *lu*, *scaleperm*, *solve*, *stat*, *options* – SuperLU_DIST object pointers.
- *sunctx* – the `SUNContext` object (see §4.1)

Return value: If successful, a `SUNLinearSolver` object; otherwise this routine will return `NULL`.

Notes: This routine analyzes the input matrix and vector to determine the linear system size and to assess the compatibility with the SuperLU_DIST library.

This routine will perform consistency checks to ensure that it is called with consistent N_Vector and SUNMatrix implementations. These are currently limited to the SUNMatrix_SLUNRloc matrix type and the NVECTOR_SERIAL, NVECTOR_OPENMP, NVECTOR_PTHREADS, NVECTOR_PARALLEL, and NVECTOR_PARHYP 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.

realtype **SUNLinSol_SuperLUDIST_GetBerr**(*SUNLinearSolver* LS)

This function returns the componentwise relative backward error of the computed solution. It takes one argument, the `SUNLinearSolver` object. The return type is `realtype`.

gridinfo_t ***SUNLinSol_SuperLUDIST_GetGridinfo**(*SUNLinearSolver* LS)

This function returns a pointer to the SuperLU_DIST structure that contains the 2D process grid. It takes one argument, the `SUNLinearSolver` object.

xLUstruct_t ***SUNLinSol_SuperLUDIST_GetLUstruct**(*SUNLinearSolver* LS)

This function returns a pointer to the SuperLU_DIST structure that contains the distributed `L` and `U` structures. It takes one argument, the `SUNLinearSolver` object.

superlu_dist_options_t ***SUNLinSol_SuperLUDIST_GetSuperLUOptions**(*SUNLinearSolver* LS)*
This function returns a pointer to the SuperLU_DIST structure that contains the options which control how the linear system is factorized and solved. It takes one argument, the `SUNLinearSolver` object.

xScalePermstruct_t ***SUNLinSol_SuperLUDIST_GetScalePermstruct**(*SUNLinearSolver* LS)*
This function returns a pointer to the SuperLU_DIST structure that contains the vectors that describe the transformations done to the matrix `A`. It takes one argument, the `SUNLinearSolver` object.

xSOLVEstruct_t ***SUNLinSol_SuperLUDIST_GetSOLVEstruct**(*SUNLinearSolver* LS)*
This function returns a pointer to the SuperLU_DIST structure that contains information for communication during the solution phase. It takes one argument the `SUNLinearSolver` object.

SuperLUStat_t ***SUNLinSol_SuperLUDIST_GetSuperLUStat**(*SUNLinearSolver* LS)*
This function returns a pointer to the SuperLU_DIST structure that stores information about runtime and flop count. It takes one argument, the `SUNLinearSolver` object.

8.15.2 SUNLinSol_SuperLUDIST Description

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;
  realtype                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 internal function evaluations,
- `berr` – the componentwise relative backward error of the computed solution,
- `grid` – pointer to the SuperLU_DIST structure that strores the 2D process grid
- `lu` – pointer to the SuperLU_DIST structure that stores the distributed `L` and `U` factors,
- `scaleperm` – pointer to the SuperLU_DIST structure that stores vectors describing the transformations done to the matrix `A`,
- `options` – pointer to the SuperLU_DIST stucture which contains options that control how the linear system is factorized and solved,
- `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.

The SUNLinSol_SuperLUDIST module is a SUNLinearSolver adapter for the SuperLU_DIST sparse matrix factorization and solver library written by X. Sherry Li and collaborators [17, 25, 26, 41]. 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 §10.1.4 for details). Additionally, the wrapper 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 factorized and 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.

The SUNLinSol_SuperLUDIST module defines implementations of all "direct" linear solver operations listed in §8.1:

- `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`

8.16 The SUNLinSol_SuperLUMT Module

The SUNLinSol_SuperLUMT implementation of the `SUNLinearSolver` class interfaces with the SuperLU_MT library. This is designed to be used with the corresponding SUNMATRIX_SPARSE matrix type, and one of the serial or shared-memory `N_Vector` 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 SuperLU_MT library has also been compiled with OpenMP.

8.16.1 SUNLinSol_SuperLUMT Usage

The header file to be included when using this module is `sunlinsol/sunlinsol.SuperLUMT.h`. The installed module library to link to is `libsundials_sunlinsolsuperlumt` *.lib* where *.lib* is typically `.so` for shared libraries and `.a` for static libraries.

The module SUNLinSol_SuperLUMT provides the following user-callable routines:

SUNLinearSolver **SUNLinSol_SuperLUMT**(*N_Vector* y, *SUNMatrix* A, int num_threads, *SUNContext* sunctx)
This constructor function creates and allocates memory for a SUNLinSol_SuperLUMT object.

Arguments:

- *y* – a template vector.
- *A* – a template matrix
- *num_threads* – desired number of threads (OpenMP or Pthreads, depending on how SuperLU_MT was installed) to use during the factorization steps.
- *sunctx* – the `SUNContext` object (see §4.1)

Return value: If successful, a `SUNLinearSolver` object; otherwise 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_MT library.

This routine will perform consistency checks to ensure that it is called with consistent `N_Vector` and `SUNMatrix` implementations. These are currently limited to the SUNMATRIX_SPARSE matrix type (using either CSR or CSC storage formats) and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

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

int **SUNLinSol_SuperLUMTSetOrdering**(*SUNLinearSolver* S, int ordering_choice)
This function sets the ordering used by SuperLU_MT for reducing fill in the linear solve.

Arguments:

- *S* – the SUNLinSol_SuperLUMT object to update.
- *ordering_choice*:
  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:

- SUNLS_SUCCESS – option successfully set
- SUNLS_MEM_NULL – S is NULL
- SUNLS_ILL_INPUT – invalid ordering_choice

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

SUNLinearSolver **SUNSuperLUMT**(*N_Vector* y, *SUNMatrix* A, int num_threads)
Wrapper for *SUNLinSol_SuperLUMT()*.

and

int **SUNSuperLUMTSetOrdering**(*SUNLinearSolver* S, int ordering_choice)
Wrapper for *SUNLinSol_SuperLUMTSetOrdering()*.

8.16.2 SUNLinSol_SuperLUMT Description

The SUNLinSol_SuperLUMT module defines the *content* field of a SUNLinearSolver to be 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 SuperLU_MT options structure.

The SUNLinSol_SuperLUMT module is a SUNLinearSolver wrapper for the SuperLU_MT sparse matrix factorization and solver library written by X. Sherry Li and collaborators [11, 24, 42]. 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 SuperLU_MT, it is assumed

that SuperLU_MT has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with SuperLU_MT (see §10.1.4 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 §5.2 for details). Moreover, since the SuperLU_MT library may be installed to support either 32-bit or 64-bit integers, it is assumed that the SuperLU_MT library is installed using the same integer precision as the SUNDIALS *sunindextype* option.

The SuperLU_MT 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 SuperLU_MT data structures. We note that in this solve SuperLU_MT operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

The SUNLinSol_SuperLUMT module defines implementations of all "direct" linear solver operations listed in §8.1:

- `SUNLinSolGetType_SuperLUMT`
- `SUNLinSolInitialize_SuperLUMT` – this sets the `first_factorize` flag to 1 and resets the internal SuperLU_MT statistics variables.
- `SUNLinSolSetup_SuperLUMT` – this performs either a LU factorization or refactorization of the input matrix.
- `SUNLinSolSolve_SuperLUMT` – this calls the appropriate SuperLU_MT 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 SuperLU_MT documentation.
- `SUNLinSolFree_SuperLUMT`

8.17 The SUNLinSol_cuSolverSp_batchQR Module

The SUNLinSol_cuSolverSp_batchQR implementation of the `SUNLinearSolver` class 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.

Warning: The SUNLinearSolver_cuSolverSp_batchQR module is experimental and subject to change.

8.17.1 SUNLinSol_cuSolverSp_batchQR description

The SUNLinearSolver_cuSolverSp_batchQR implementation provides an interface to the batched sparse QR factorization method provided by the NVIDIA cuSOLVER library [39]. 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`.

8.17.2 SUNLinSol_cuSolverSp_batchQR functions

The `SUNLinearSolver_cuSolverSp_batchQR` module defines implementations of all "direct" linear solver operations listed in §8.1:

- `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:

SUNLinearSolver **SUNLinSol_cuSolverSp_batchQR**(*N_Vector* y, *SUNMatrix* A, cusolverHandle_t cusol, *SUNContext* sunctx)

The function `SUNLinSol_cuSolverSp_batchQR` creates and allocates memory for a SUNLinearSolver object.

Arguments:

- *y* – a vector for checking compatibility with the solver.
- *A* – a SUNMATRIX_cuSparse matrix for checking compatibility with the solver.
- *cusol* – cuSolverSp object to use.
- *sunctx* – the *SUNContext* object (see §4.1)

Return value: If successful, 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 `N_Vector` and `SUNMatrix` implementations. These are currently limited to the SUNMATRIX_CUSPARSE matrix type and the NVECTOR_CUDA vector type. Since the SUNMATRIX_CUSPARSE matrix type is only compatible with the NVECTOR_CUDA the restriction is also in place for the linear solver. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

void **SUNLinSol_cuSolverSp_batchQR_GetDescription**(*SUNLinearSolver* LS, char **desc)

The function `SUNLinSol_cuSolverSp_batchQR_GetDescription` accesses the string description of the object (empty by default).

void **SUNLinSol_cuSolverSp_batchQR_SetDescription**(*SUNLinearSolver* LS, const char *desc)
The function SUNLinSol_cuSolverSp_batchQR_SetDescription sets the string description of the object (empty by default).

void **SUNLinSol_cuSolverSp_batchQR_GetDeviceSpace**(*SUNLinearSolver* S, size_t *cuSolverInternal, size_t *cuSolverWorkspace)
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 agrument 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.

8.17.3 SUNLinSol_cuSolverSp_batchQR content

The SUNLinSol_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 buffer for Q and R       */
   size_t             workspace_size;  /* size of cusolver memory for 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         */
};
```

8.18 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 modules. 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_SUNLinSolGetID: Verifies the returned solver identifier 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, N_Vector 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_SUNLinSolSetZeroGuess` (iterative solvers only): Verifies that `SUNLinSolSetZeroGuess` 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 9

Tools for Memory Management

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.

9.1 The SUNMemoryHelper API

This API consists of three new SUNDIALS types: *SUNMemoryType*, *SUNMemory*, and *SUNMemoryHelper*:

typedef struct _SUNMemory ***SUNMemory**

The SUNMemory type is a pointer a structure containing a pointer to actual data (ptr), the data memory type, and a flag indicating ownership of that data pointer. This structure is defined as

```
struct _SUNMemory
{
  void*         ptr;
  SUNMemoryType type;
  booleantype   own;
};
```

enum **SUNMemoryType**

The SUNMemoryType type is an enumeration that defines the supported memory types:

```
typedef enum
{
  SUNMEMTYPE_HOST,      /* pageable memory accessible on the host     */
  SUNMEMTYPE_PINNED,    /* page-locked memory accesible on the host   */
  SUNMEMTYPE_DEVICE,    /* memory accessible from the device          */
  SUNMEMTYPE_UVM        /* memory accessible from the host or device  */
} SUNMemoryType;
```

typedef struct _SUNMemoryHelper ***SUNMemoryHelper**

The SUNMemoryHelper type is a pointer to a structure containing a pointer to the implementation-specific member data (content) and a virtual method table of member functions (ops). This strucutre is defined as

```
struct _SUNMemoryHelper
{
```

(continues on next page)

(continued from previous page)

```
  void*               content;
  SUNMemoryHelper_Ops ops;
};
```

typedef struct _SUNMemoryHelper_Ops ***SUNMemoryHelper_Ops**

The `SUNMemoryHelper_Ops` type is defined as a pointer to the structure containing the function pointers to the member function implementations. This structure is define as

```
struct _SUNMemoryHelper_Ops
{
  /* operations that implementations are required to provide */
  int (*alloc)(SUNMemoryHelper, SUNMemory* memptr size_t mem_size,
               SUNMemoryType mem_type, void* queue);
  int (*dealloc)(SUNMemoryHelper, SUNMemory mem, void* queue);
  int (*copy)(SUNMemoryHelper, SUNMemory dst, SUNMemory src,
              size_t mem_size, void* queue);

  /* operations that provide default implementations */
  int             (*copyasync)(SUNMemoryHelper, SUNMemory dst,
                               SUNMemory src, size_t mem_size,
                               void* queue);
  SUNMemoryHelper (*clone)(SUNMemoryHelper);
  int             (*destroy)(SUNMemoryHelper);
};
```

9.1.1 Implementation defined operations

The SUNMemory API defines the following operations that an implementation to must define:

SUNMemory **SUNMemoryHelper_Alloc**(*SUNMemoryHelper* helper, *SUNMemory* *memptr, size_t mem_size, *SUNMemoryType* mem_type, void *queue)

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` – the `SUNMemoryHelper` object.
- `memptr` – pointer to the allocated `SUNMemory`.
- `mem_size` – the size in bytes of the `ptr`.
- `mem_type` – the `SUNMemoryType` of the `ptr`.
- `queue` – typically a handle for an object representing an alternate execution stream (e.g., a CUDA/HIP stream or SYCL queue), but it can also be any implementation specific data.

Returns:

- An `int` flag indicating success (zero) or failure (non-zero).

int **SUNMemoryHelper_Dealloc**(*SUNMemoryHelper* helper, *SUNMemory* mem, void *queue)

Deallocates the `mem->ptr` field if it is owned by `mem`, and then deallocates the `mem` object.

Arguments:

- `helper` – the `SUNMemoryHelper` object.

- `mem` – the `SUNMemory` object.
- `queue` – typically a handle for an object representing an alternate execution stream (e.g., a CUDA/HIP stream or SYCL queue), but it can also be any implementation specific data.

Returns:

- An `int` flag indicating success (zero) or failure (non-zero).

int **SUNMemoryHelper_Copy**(*SUNMemoryHelper* helper, *SUNMemory* dst, *SUNMemory* src, size_t mem_size, void *queue)

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` – the `SUNMemoryHelper` object.
- `dst` – the destination memory to copy to.
- `src` – the source memory to copy from.
- `mem_size` – the number of bytes to copy.
- `queue` – typically a handle for an object representing an alternate execution stream (e.g., a CUDA/HIP stream or SYCL queue), but it can also be any implementation specific data.

Returns:

- An `int` flag indicating success (zero) or failure (non-zero).

9.1.2 Utility Functions

The SUNMemoryHelper API defines the following functions which do not require a SUNMemoryHelper instance:

SUNMemory **SUNMemoryHelper_Alias**(*SUNMemory* mem1)

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` – a `SUNMemory` object.

Returns:

- A `SUNMemory` object or `NULL` if an error occurs.

SUNMemory **SUNMemoryHelper_Wrap**(void *ptr, *SUNMemoryType* mem_type)

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` – the data pointer to wrap in a `SUNMemory` object.
- `mem_type` – the `SUNMemoryType` of the `ptr`.

Returns:

- A `SUNMemory` object or `NULL` if an error occurs.

SUNMemoryHelper **SUNMemoryHelper_NewEmpty**()

Returns an empty `SUNMemoryHelper`. This is useful for building custom `SUNMemoryHelper` implementations.

Returns:

- A SUNMemoryHelper object or NULL if an error occurs.

int **SUNMemoryHelper_CopyOps**(*SUNMemoryHelper* src, *SUNMemoryHelper* dst)

Copies the ops field of src to the ops field of dst. This is useful for building custom SUNMemoryHelper implementations.

Arguments:

- src – the object to copy from.
- dst – the object to copy to.

Returns:

- An int flag indicating success (zero) or failure (non-zero).

9.1.3 Implementation overridable operations with defaults

In addition, the SUNMemoryHelper API defines the following *optionally overridable* operations which an implementation may define:

int **SUNMemoryHelper_CopyAsync**(*SUNMemoryHelper* helper, *SUNMemory* dst, *SUNMemory* src, size_t mem_size, void *queue)

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 – the SUNMemoryHelper object.
- dst – the destination memory to copy to.
- src – the source memory to copy from.
- mem_size – the number of bytes to copy.
- queue – typically a handle for an object representing an alternate execution stream (e.g., a CUDA/HIP stream or SYCL queue), but it can also be any implementation specific data.

Returns:

An int flag indicating success (zero) or failure (non-zero).

Note: If this operation is not defined by the implementation, then *SUNMemoryHelper_Copy()* will be used.

SUNMemoryHelper **SUNMemoryHelper_Clone**(*SUNMemoryHelper* helper)

Clones the SUNMemoryHelper object itself.

Arguments:

- helper – the SUNMemoryHelper object to clone.

Returns:

- A SUNMemoryHelper object.

Note: 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.

int **SUNMemoryHelper_Destroy**(*SUNMemoryHelper* helper)
Destroys (frees) the `SUNMemoryHelper` object itself.

Arguments:

- `helper` – the `SUNMemoryHelper` object to destroy.

Returns:

- An `int` flag indicating success (zero) or failure (non-zero).

Note: 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.

9.1.4 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`.

9.2 The SUNMemoryHelper_Cuda Implementation

The SUNMemoryHelper_Cuda module is an implementation of the `SUNMemoryHelper` API that interfaces to the NVIDIA [38] library. The implementation defines the constructor

SUNMemoryHelper **SUNMemoryHelper_Cuda**(*SUNContext* sunctx)
Allocates and returns a `SUNMemoryHelper` object for handling CUDA memory if successful. Otherwise it returns `NULL`.

9.2.1 SUNMemoryHelper_Cuda API Functions

The implementation provides the following operations defined by the `SUNMemoryHelper` API:

SUNMemory **SUNMemoryHelper_Alloc_Cuda**(*SUNMemoryHelper* helper, *SUNMemory* memptr, size_t mem_size, *SUNMemoryType* mem_type, void *queue)
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` – the `SUNMemoryHelper` object.
- `memptr` – pointer to the allocated `SUNMemory`.

- mem_size – the size in bytes of the ptr.
- mem_type – the SUNMemoryType of the ptr. Supported values 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.
- queue – currently unused.

Returns:

- An int flag indicating success (zero) or failure (non-zero).

int **SUNMemoryHelper_Dealloc_Cuda**(*SUNMemoryHelper* helper, *SUNMemory* mem, void *queue)

Deallocates the mem->ptr field if it is owned by mem, and then deallocates the mem object.

Arguments:

- helper – the SUNMemoryHelper object.
- mem – the SUNMemory object.
- queue – currently unused.

Returns:

- An int flag indicating success (zero) or failure (non-zero).

int **SUNMemoryHelper_Copy_Cuda**(*SUNMemoryHelper* helper, *SUNMemory* dst, *SUNMemory* src, size_t mem_size, void *queue)

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:

- helper – the SUNMemoryHelper object.
- dst – the destination memory to copy to.
- src – the source memory to copy from.
- mem_size – the number of bytes to copy.
- queue – currently unused.

Returns:

- An int flag indicating success (zero) or failure (non-zero).

int **SUNMemoryHelper_CopyAsync_Cuda**(*SUNMemoryHelper* helper, *SUNMemory* dst, *SUNMemory* src, size_t mem_size, void *queue)

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:

- helper – the SUNMemoryHelper object.
- dst – the destination memory to copy to.
- src – the source memory to copy from.

- `mem_size` – the number of bytes to copy.
- `queue` – the `cudaStream_t` handle for the stream that the copy will be performed on.

Returns:

- An `int` flag indicating success (zero) or failure (non-zero).

9.3 The SUNMemoryHelper_Hip Implementation

The SUNMemoryHelper_Hip module is an implementation of the `SUNMemoryHelper` API that interfaces to the AMD ROCm HIP library [35]. The implementation defines the constructor

SUNMemoryHelper **SUNMemoryHelper_Hip**(*SUNContext* sunctx)
Allocates and returns a `SUNMemoryHelper` object for handling HIP memory if successful. Otherwise it returns `NULL`.

9.3.1 SUNMemoryHelper_Hip API Functions

The implementation provides the following operations defined by the `SUNMemoryHelper` API:

SUNMemory **SUNMemoryHelper_Alloc_Hip**(*SUNMemoryHelper* helper, *SUNMemory* memptr, size_t mem_size, *SUNMemoryType* mem_type, void *queue)
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` – the `SUNMemoryHelper` object.
- `memptr` – pointer to the allocated `SUNMemory`.
- `mem_size` – the size in bytes of the `ptr`.
- `mem_type` – the `SUNMemoryType` of the `ptr`. Supported values 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`.
- `queue` – currently unused.

Returns:

- An `int` flag indicating success (zero) or failure (non-zero).

int **SUNMemoryHelper_Dealloc_Hip**(*SUNMemoryHelper* helper, *SUNMemory* mem, void *queue)
Deallocates the `mem->ptr` field if it is owned by `mem`, and then deallocates the `mem` object.

Arguments:

- `helper` – the `SUNMemoryHelper` object.
- `mem` – the `SUNMemory` object.

Returns:

- An `int` flag indicating success (zero) or failure (non-zero).

int **SUNMemoryHelper_Copy_Hip**(*SUNMemoryHelper* helper, *SUNMemory* dst, *SUNMemory* src, size_t mem_size, void *queue)

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:

- `helper` – the `SUNMemoryHelper` object.
- `dst` – the destination memory to copy to.
- `src` – the source memory to copy from.
- `mem_size` – the number of bytes to copy.

Returns:

- An `int` flag indicating success (zero) or failure (non-zero).

int **SUNMemoryHelper_CopyAsync_Hip**(*SUNMemoryHelper* helper, *SUNMemory* dst, *SUNMemory* src, size_t mem_size, void *queue)

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:

- `helper` – the `SUNMemoryHelper` object.
- `dst` – the destination memory to copy to.
- `src` – the source memory to copy from.
- `mem_size` – the number of bytes to copy.
- `queue` – the `hipStream_t` handle for the stream that the copy will be performed on.

Returns:

- An `int` flag indicating success (zero) or failure (non-zero).

9.4 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 **SUNMemoryHelper_Sycl**(*SUNContext* sunctx)

Allocates and returns a `SUNMemoryHelper` object for handling SYCL memory using the provided queue. Otherwise it returns `NULL`.

9.4.1 SUNMemoryHelper_Sycl API Functions

The implementation provides the following operations defined by the `SUNMemoryHelper` API:

SUNMemory **SUNMemoryHelper_Alloc_Sycl**(*SUNMemoryHelper* helper, *SUNMemory* memptr, size_t mem_size, *SUNMemoryType* mem_type, void *queue)

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` – the `SUNMemoryHelper` object.
- `memptr` – pointer to the allocated `SUNMemory`.
- `mem_size` – the size in bytes of the `ptr`.
- `mem_type` – the `SUNMemoryType` of the `ptr`. Supported values 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`.
- `queue` – the `sycl::queue` handle for the stream that the allocation will be performed on.

Returns:

- An `int` flag indicating success (zero) or failure (non-zero).

int **SUNMemoryHelper_Dealloc_Sycl**(*SUNMemoryHelper* helper, *SUNMemory* mem, void *queue)

Deallocates the `mem->ptr` field if it is owned by `mem`, and then deallocates the `mem` object.

Arguments:

- `helper` – the `SUNMemoryHelper` object.
- `mem` – the `SUNMemory` object.
- `queue` – the `sycl::queue` handle for the queue that the deallocation will be performed on.

Returns:

- An `int` flag indicating success (zero) or failure (non-zero).

int **SUNMemoryHelper_Copy_Sycl**(*SUNMemoryHelper* helper, *SUNMemory* dst, *SUNMemory* src, size_t mem_size, void *queue)

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:

- `helper` – the `SUNMemoryHelper` object.
- `dst` – the destination memory to copy to.
- `src` – the source memory to copy from.
- `mem_size` – the number of bytes to copy.
- `queue` – the `sycl::queue` handle for the queue that the copy will be performed on.

Returns:

- An `int` flag indicating success (zero) or failure (non-zero).

int **SUNMemoryHelper_CopyAsync_Sycl**(*SUNMemoryHelper* helper, *SUNMemory* dst, *SUNMemory* src, size_t mem_size, void *queue)

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:

- `helper` – the `SUNMemoryHelper` object.
- `dst` – the destination memory to copy to.
- `src` – the source memory to copy from.
- `mem_size` – the number of bytes to copy.
- `queue` – the `sycl::queue` handle for the queue that the copy will be performed on.

Returns:

- An `int` flag indicating success (zero) or failure (non-zero).

Chapter 10

SUNDIALS 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 -zxf 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 begin with a few common observations:

1. The remainder of this chapter will follow these conventions:

   `SOLVERDIR` is the directory `SOLVER-X.Y.Z` created above; i.e. the directory containing the SUNDIALS sources.

   `BUILDDIR` is the (temporary) directory under which SUNDIALS is built.

   `INSTDIR` is the directory under which the SUNDIALS exported header files and libraries will be installed. Typically, header files are exported under a directory `INSTDIR/include` while libraries are installed under `INSTDIR/lib`, with `INSTDIR` specified at configuration time.

2. 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.

3. The installation directory `INSTDIR` can not be the same as the source directory `SOLVERDIR`.

4. 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.)

Further details on the CMake-based installation procedures, instructions for manual compilation, and a roadmap of the resulting installed libraries and exported header files, are provided in §10.1 and §10.2.

10.1 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 3.12.0 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` or `cmake-gui`), while on Windows it requires Visual Studio. While many Linux distributions offer CMake, the version included may be out of date. CMake adds new features regularly, and you should download the latest version from http://www.cmake.org. Build instructions for CMake (only necessary for Unix-like systems) can be found on the CMake website. Once CMake is installed, Linux/Unix users will be able to use `ccmake` or `cmake-gui` (depending on the version of CMake), 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.

10.1.1 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, or from a wxWidgets or QT based GUI by using the `cmake-gui` command. Examples for using both text and graphical 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 (...)/INSTDIR
$ mkdir (...)/BUILDDIR
$ cd (...)/BUILDDIR
```

10.1.1.1 Building with the GUI

Using CMake with the `ccmake` GUI follows the general process:

1. Select and modify values, run configure (`c` key)
2. New values are denoted with an asterisk
3. 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

4. Repeat until all values are set as desired and the generate option is available (`g` key)
5. Some variables (advanced variables) are not visible right away; to see advanced variables, toggle to advanced mode (`t` key)
6. To search for a variable press the `/` key, and to repeat the search, press the `n` key

Using CMake with the `cmake-gui` GUI follows a similar process:

1. Select and modify values, click `Configure`
2. The first time you click `Configure`, make sure to pick the appropriate generator (the following will assume generation of Unix Makfiles).
3. New values are highlighted in red
4. To set a variable, click on or move the cursor to the variable and press enter
   - If it is a boolean (`ON/OFF`) it will check/uncheck the box
   - If it is string or file, it will allow editing of the string. Additionally, an ellipsis button will appear `...` on the far right of the entry. Clicking this button will bring up the file or directory selection dialog.
   - For files and directories, the `<tab>` key can be used to complete
5. Repeat until all values are set as desired and click the `Generate` button
6. Some variables (advanced variables) are not visible right away; to see advanced variables, click the `advanced` button

To build the default configuration using the curses GUI, from the `BUILDDIR` enter the `ccmake` command and point to the `SOLVERDIR`:

```
$ ccmake (...)/SOLVERDIR
```

Similarly, to build the default configuration using the wxWidgets GUI, from the `BUILDDIR` enter the `cmake-gui` command and point to the `SOLVERDIR`:

```
$ cmake-gui (...)/SOLVERDIR
```

The default curses configuration screen is shown in the following figure.

The default INSTDIR for both SUNDIALS and the corresponding examples can be changed by setting the `CMAKE_INSTALL_PREFIX` and the `EXAMPLES_INSTALL_PATH` as shown in the following figure.

Pressing the `g` key or clicking `generate` 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
```

or for a faster parallel build (e.g. using 4 threads), you can run

```
$ make -j 4
```

To install SUNDIALS in the installation directory specified in the configuration, simply run:

```
$ make install
```

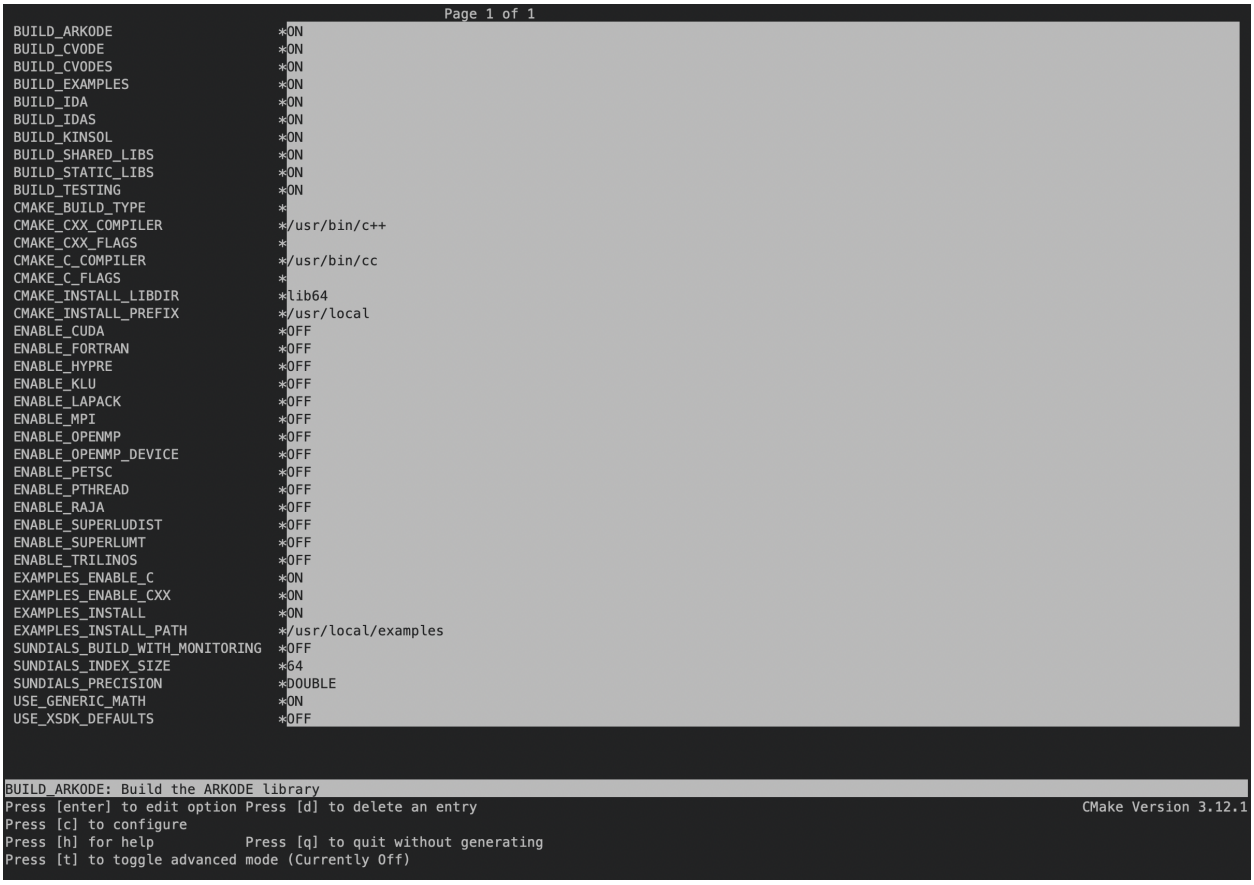

Fig. 10.1: Default configuration screen. Note: Initial screen is empty. To get this default configuration, press 'c' repeatedly (accepting default values denoted with asterisk) until the 'g' option is available.

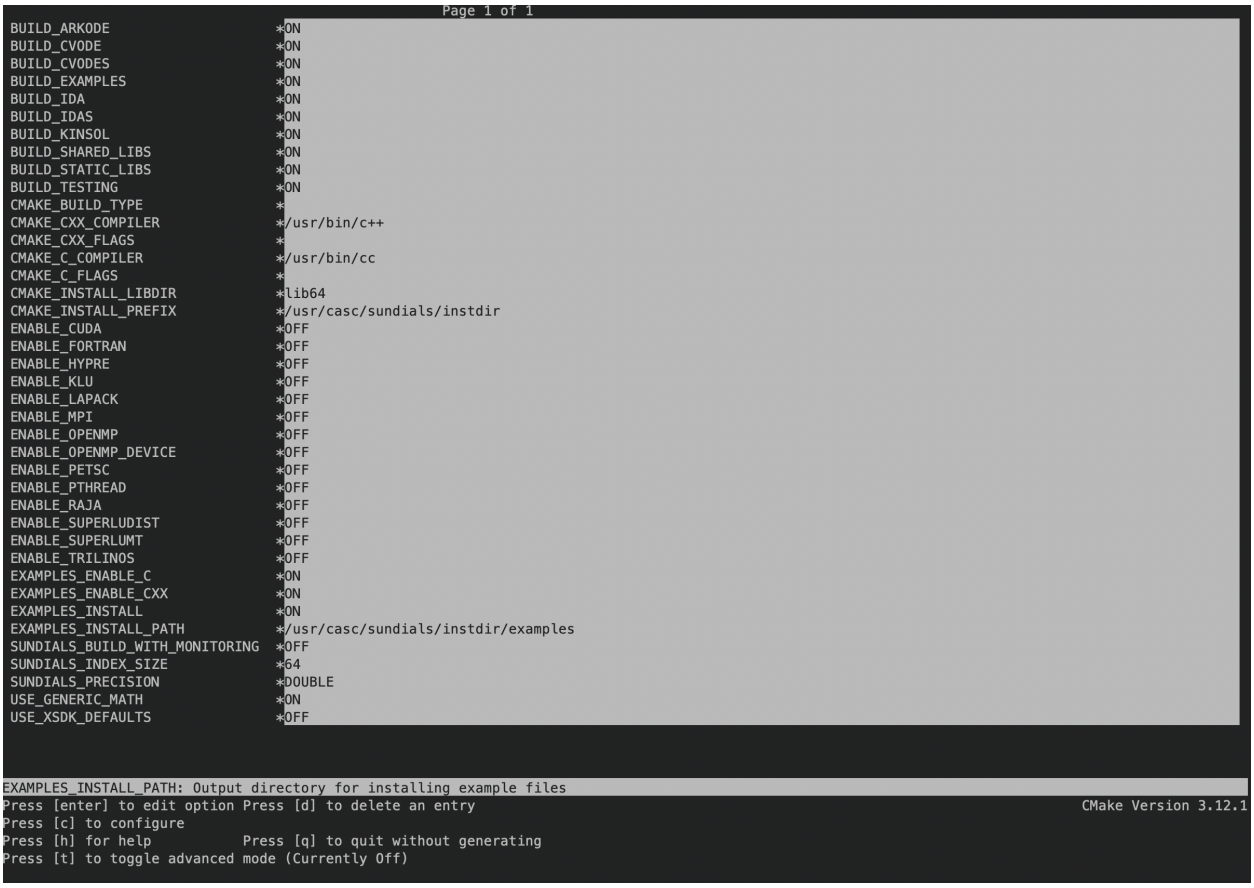

Fig. 10.2: Changing the INSTDIR for SUNDIALS and corresponding EXAMPLES.

10.1.1.2 Building from the command line

Using CMake from the command line is simply a matter of specifying CMake variable settings with the `cmake` command. The following will build the default configuration:

```
$ cmake -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
>  -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
>  ../srcdir
$ make
$ make install
```

10.1.2 Configuration options (Unix/Linux)

A complete list of all available options for a CMake-based SUNDIALS configuration is provide below. Note that the default values shown are for a typical configuration on a Linux system and are provided as illustration only.

BUILD_ARKODE
Build the ARKODE library

Default: `ON`

BUILD_CVODE
Build the CVODE library

Default: `ON`

BUILD_CVODES
Build the CVODES library

Default: `ON`

BUILD_IDA
Build the IDA library

Default: `ON`

BUILD_IDAS
Build the IDAS library

Default: `ON`

BUILD_KINSOL
Build the KINSOL library

Default: `ON`

BUILD_SHARED_LIBS
Build shared libraries

Default: `ON`

BUILD_STATIC_LIBS
Build static libraries

Default: `ON`

CMAKE_BUILD_TYPE
Choose the type of build, options are: `None`, `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_C_STANDARD
The C standard to build C parts of SUNDIALS with.

Default: 99

Options: 90, 99, 11, 17.

CMAKE_C_EXTENSIONS
Enable compiler specific C extensions.

Default: `OFF`

CMAKE_CXX_COMPILER
C++ compiler

Default: `/usr/bin/c++`

Note: A C++ compiler is only required when a feature requiring C++ is enabled (e.g., CUDA, HIP, SYCL, RAJA, etc.) or the C++ examples are enabled.

All SUNDIALS solvers can be used from C++ applications 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

Options: 98, 11, 14, 17, 20.

CMAKE_CXX_EXTENSIONS
Enable compiler specific C++ extensions.

Default: `OFF`

CMAKE_Fortran_COMPILER
Fortran compiler

Default: `/usr/bin/gfortran`

Note: Fortran support (and all related options) are triggered only if either Fortran-C support (`BUILD_FORTRAN_MODULE_INTERFACE`) or LAPACK (`ENABLE_LAPACK`) support is enabled.

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_LIBDIR
The directory under which libraries will be installed.

Default: Set based on the system: `lib`, `lib64`, or `lib/<multiarch-tuple>`

CMAKE_INSTALL_PREFIX
Install path prefix, prepended onto install directories

Default: `/usr/local`

Note: The user must have write access to the location specified through this option. Exported SUNDIALS header files and libraries will be installed under subdirectories `include` and `lib` of `CMAKE_INSTALL_PREFIX`,

respectively.

ENABLE_CUDA
Build the SUNDIALS CUDA modules.

Default: OFF

CMAKE_CUDA_ARCHITECTURES
Specifies the CUDA architecture to compile for.

Default: sm_30

ENABLE_XBRAID
Enable or disable the ARKStep + XBraid interface.

Default: OFF

Note: See additional information on building with *XBraid* enabled in §10.1.4.

EXAMPLES_ENABLE_C
Build the SUNDIALS C examples

Default: ON

EXAMPLES_ENABLE_CXX
Build the SUNDIALS C++ examples

Default: OFF

EXAMPLES_ENABLE_CUDA
Build the SUNDIALS CUDA examples

Default: OFF

Note: You need to enable CUDA support to build these examples.

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`.

BUILD_FORTRAN_MODULE_INTERFACE
Enable Fortran2003 interface

Default: `OFF`

ENABLE_HYPRE
Flag to enable *hypre* support

Default: `OFF`

Note: See additional information on building with *hypre* enabled in §10.1.4.

HYPRE_INCLUDE_DIR
Path to *hypre* header files

Default: none

HYPRE_LIBRARY
Path to *hypre* installed library files

Default: none

ENABLE_KLU
Enable KLU support

Default: `OFF`

Note: See additional information on building with KLU enabled in §10.1.4.

KLU_INCLUDE_DIR
Path to SuiteSparse header files

Default: none

KLU_LIBRARY_DIR
Path to SuiteSparse installed library files

Default: none

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 §10.1.4.

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
`mpif90` program

Default:

Note: This option is triggered only if MPI is enabled (`ENABLE_MPI` is `ON`) and Fortran-C support is enabled (`EXAMPLES_ENABLE_F2003` 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`

ONEMKL_DIR
Path to oneMKL installation.

Default: none

ENABLE_OPENMP
Enable OpenMP support (build the OpenMP NVector)

Default: `OFF`

ENABLE_PETSC
Enable PETSc support

Default: `OFF`

Note: See additional information on building with PETSc enabled in §10.1.4.

PETSC_DIR
Path to PETSc installation

Default: none

PETSC_LIBRARIES
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: none

PETSC_INCLUDES
Semi-colon separated list of PETSc include directroies. Unless provided by the user, this is autopopulated based on the PETSc installation found in `PETSC_DIR`.

Default: none

ENABLE_PTHREAD
Enable Pthreads support (build the Pthreads NVector)

Default: `OFF`

ENABLE_RAJA
Enable RAJA support.

Default: OFF

Note: You need to enable CUDA or HIP 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 wtih SuperLU_DIST enabled in §10.1.4.

SUPERLUDIST_INCLUDE_DIR
Path to SuperLU_DIST header files (under a typical SuperLU_DIST install, this is typically the SuperLU_DIST SRC directory)

Default: none

SUPERLUDIST_LIBRARY_DIR
Path to SuperLU_DIST installed library files

Default: none

SUPERLUDIST_LIBRARIES
Semi-colon separated list of libraries needed for SuperLU_DIST

Default: none

SUPERLUDIST_OpenMP
Enable SUNDIALS support for SuperLU_DIST built with OpenMP

Default: none

Note: SuperLU_DIST must be built with OpenMP support for this option to function. Additionally the environment variable OMP_NUM_THREADS must be set to the desired number of threads.

ENABLE_SUPERLUMT
Enable SuperLU_MT support

Default: OFF

Note: See additional information on building with SuperLU_MT enabled in §10.1.4.

SUPERLUMT_INCLUDE_DIR
Path to SuperLU_MT header files (under a typical SuperLU_MT install, this is typically the SuperLU_MT SRC directory)

Default: none

SUPERLUMT_LIBRARY_DIR
Path to SuperLU_MT installed library files

Default: none

SUPERLUMT_THREAD_TYPE
Must be set to Pthread or OpenMP, depending on how SuperLU_MT was compiled.

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., `dpcpp` in order to build with SYCL support.

SUNDIALS_BUILD_WITH_MONITORING
Build SUNDIALS with capabilties for fine-grained monitoring of solver progress and statistics. This is primarily useful for debugging.

Default: OFF

Warning: Building with monitoring may result in minor performance degradation even if monitoring is not utilized.

SUNDIALS_BUILD_WITH_PROFILING
Build SUNDIALS with capabilties for fine-grained profiling.

Default: OFF

Warning: Profiling will impact performance, and should be enabled judiciously.

ENABLE_CALIPER
Enable CALIPER support

Default: OFF

Note: Using Caliper requires setting *SUNDIALS_BUILD_WITH_PROFILING* to `ON`.

CALIPER_DIR
Path to the root of a Caliper installation

Default: None

SUNDIALS_F77_FUNC_CASE
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
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
Integer type used for SUNDIALS indices. The size must match the size provided for the SUNDIALS_INDEX_SIZE option.

Default: Automatically determined based on *SUNDIALS_INDEX_SIZE*

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
The floating-point precision used in SUNDIALS packages and class implementations, 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

XBRAID_DIR
The root directory of the XBraid installation.

Default: OFF

XBRAID_INCLUDES
Semi-colon separated list of XBraid include directories. Unless provided by the user, this is autopopulated based on the XBraid installation found in XBRAID_DIR.

Default: none

XBRAID_LIBRARIES
Semi-colon separated list of XBraid link libraries. Unless provided by the user, this is autopopulated based on the XBraid installation found in XBRAID_DIR.

Default: none

USE_XSDK_DEFAULTS
Enable xSDK (see https://xsdk.info 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

10.1.3 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 `mpif90` 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 \
> /home/myname/sundials/srcdir

% make install
```

To disable installation of the examples, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DENABLE_MPI=ON \
> -DEXAMPLES_INSTALL=OFF \
> /home/myname/sundials/srcdir

% make install
```

10.1.4 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.

10.1.4.1 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/srcdir

% make install
```

Note: 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.

SUNDIALS has been tested with OpenBLAS 0.3.18.

10.1.4.2 Building with KLU

KLU is a software package for the direct solution of sparse nonsymmetric linear systems of equations that arise in circuit simulation and is part of SuiteSparse, a suite of sparse matrix software. The library is developed by Texas A&M University and is available from the SuiteSparse GitHub repository.

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`.

SUNDIALS has been tested with SuiteSparse version 5.10.1.

10.1.4.3 Building with SuperLU_DIST

SuperLU_DIST is a general purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations in a distributed memory setting. The library is developed by Lawrence Berkeley National Laboratory and is available from the SuperLU_DIST GitHub repository.

To enable SuperLU_DIST, set `ENABLE_SUPERLUDIST` to `ON`, set `SUPERLUDIST_INCLUDE_DIR` to the `SRC` path of the SuperLU_DIST installation, and set the variable `SUPERLUMT_LIBRARY_DIR` to the `lib` path of the SuperLU_DIST installation. 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` utilize the OpenMP functionality of SuperLU_DIST.

SUNDIALS has been tested with SuperLU_DIST 7.1.1.

10.1.4.4 Building with SuperLU_MT

SuperLU_MT is a general purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations on shared memory parallel machines. The library is developed by Lawrence Berkeley National Laboratory and is available from the SuperLU_MT GitHub repository.

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.

SUNDIALS has been tested with SuperLU_MT version 3.1.

10.1.4.5 Building with PETSc

The Portable, Extensible Toolkit for Scientific Computation (PETSc) is a suite of data structures and routines for simulating applications modeled by partial differential equations. The library is developed by Argonne National Laboratory and is available from the PETSc GitLab repository.

To enable PETSc, set `ENABLE_PETSC` to `ON`, and 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 PETSc libraries in `PETSC_LIBRARIES`.

SUNDIALS has been tested with PETSc version 3.16.1.

10.1.4.6 Building with *hypre*

hypre is a library of high performance preconditioners and solvers featuring multigrid methods for the solution of large, sparse linear systems of equations on massively parallel computers. The library is developed by Lawrence Livermore National Laboratory and is available from the hypre GitHub repository.

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` is compatible with `HYPRE_BigInt` in the *hypre* installation.

SUNDIALS has been tested with *hypre* version 2.23.0

10.1.4.7 Building with MAGMA

The Matrix Algebra on GPU and Multicore Architectures (MAGMA) project provides a dense linear algebra library similar to LAPACK but targeting heterogeneous architectures. The library is developed by the University of Tennessee and is available from the UTK webpage.

To enable the SUNDIALS MAGMA interface set `ENABLE_MAGMA` to `ON`, `MAGMA_DIR` to the MAGMA installation path, and `SUNDIALS_MAGMA_BACKENDS` to the desired MAGMA backend to use with SUNDIALS e.g., `CUDA` or `HIP`.

SUNDIALS has been tested with MAGMA version 2.6.1.

10.1.4.8 Building with oneMKL

The Intel oneAPI Math Kernel Library (oneMKL) includes CPU and DPC++ interfaces for LAPACK dense linear algebra routines. The SUNDIALS oneMKL interface targets the DPC++ routines, to utilize the CPU routine see §10.1.4.1.

To enable the SUNDIALS oneMKL interface set `ENABLE_ONEMKL` to `ON` and `ONEMKL_DIR` to the oneMKL installation path.

SUNDIALS has been tested with oneMKL version 2021.4.

10.1.4.9 Building with CUDA

The NVIDIA CUDA Toolkit provides a development environment for GPU-accelerated computing with NVIDIA GPUs. The CUDA Toolkit and compatible NVIDIA drivers are available from the NVIDIA developer website.

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`.

SUNDIALS has been tested with the CUDA toolkit versions 10 and 11.

10.1.4.10 Building with RAJA

RAJA is a performance portability layer developed by Lawrence Livermore National Laboratory and can be obtained from the RAJA GitHub repository.

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`.

SUNDIALS has been tested with RAJA version 0.14.0.

10.1.4.11 Building with XBraid

XBraid is parallel-in-time library implementing an optimal-scaling multigrid reduction in time (MGRIT) solver. The library is developed by Lawrence Livermore National Laboratory and is available from the XBraid GitHub repository.

To enable XBraid support, set `ENABLE_XBRAID` to `ON`, set `XBRAID_DIR` to the root install location of XBraid or the location of the clone of the XBraid repository.

Note: At this time the XBraid types `braid_Int` and `braid_Real` are hard-coded to `int` and `double` respectively. As such SUNDIALS must be configured with `SUNDIALS_INDEX_SIZE` set to `32` and `SUNDIALS_PRECISION` set to `double`. Additionally, SUNDIALS must be configured with `ENABLE_MPI` set to `ON`.

SUNDIALS has been tested with XBraid version 3.0.0.

10.1.5 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
```

10.1.6 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 or cmake-gui 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.

10.1.7 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.

10.2 Installed libraries and exported header files

Using the CMake SUNDIALS build system, the command

```
$ make install
```

will install the libraries under `LIBDIR` and the public header files under `INCLUDEDIR`. The values for these directories are `INSTDIR/lib` and `INSTDIR/include`, respectively. The location can be changed by setting the CMake variable `CMAKE_INSTALL_PREFIX`. Although all installed libraries reside under `LIBDIR/lib`, the public header files are further organized into subdirectories under `INCLUDEDIR/include`.

The installed libraries and exported header files are listed for reference in the table below. The file extension `.LIB` is typically `.so` for shared libraries and `.a` for static libraries. Note that, in this table 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., `sunlinsol_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.

10.2.1 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::cvode SUNDIALS::nvecpetsc)
```

Table 10.1: SUNDIALS shared libraries and header files

Shared	Headers	sundials/sundials_band.h
		sundials/sundials_config.h
		sundials/sundials_context.h
		sundials/sundials_cuda_policies.hpp

continues on next page

Table 10.1 – continued from previous page

		sundials/sundials_dense.h
		sundials/sundials_direct.h
		sundials/sundials_hip_policies.hpp
		sundials/sundials_iterative.h
		sundials/sundials_linearsolver.h
		sundials/sundials_math.h
		sundials/sundials_matrix.h
		sundials/sundials_memory.h
		sundials/sundials_mpi_types.h
		sundials/sundials_nonlinearsolver.h
		sundials/sundials_nvector.h
		sundials/sundials_types.h
		sundials/sundials_version.h
		sundials/sundials_xbraid.h
NVECTOR Modules		
SERIAL	Libraries	libsundials_nvecserial.LIB
	Headers	nvector/nvector_serial.h
PARALLEL	Libraries	libsundials_nvecparallel.LIB
	Headers	nvector/nvector_parallel.h
OPENMP	Libraries	libsundials_nvecopenmp.LIB
	Headers	nvector/nvector_openmp.h
PTHREADS	Libraries	libsundials_nvecpthreads.LIB
	Headers	nvector/nvector_pthreads.h
PARHYP	Libraries	libsundials_nvecparhyp.LIB
	Headers	nvector/nvector_parhyp.h
PETSC	Libraries	libsundials_nvecpetsc.LIB
	Headers	nvector/nvector_petsc.h
CUDA	Libraries	libsundials_nveccuda.LIB
	Headers	nvector/nvector_cuda.h
HIP	Libraries	libsundials_nvechip.LIB
	Headers	nvector/nvector_hip.h
RAJA	Libraries	libsundials_nveccudaraja.LIB
		libsundials_nvechipraja.LIB
	Headers	nvector/nvector_raja.h
SYCL	Libraries	libsundials_nvecsycl.LIB
	Headers	nvector/nvector_sycl.h
MANYVECTOR	Libraries	libsundials_nvecmanyvector.LIB
	Headers	nvector/nvector_manyvector.h
MPIMANYVECTOR	Libraries	libsundials_nvecmpimanyvector.LIB
	Headers	nvector/nvector_mpimanyvector.h
MPIPLUSX	Libraries	libsundials_nvecmpiplusx.LIB
	Headers	nvector/nvector_mpiplusx.h
SUNMATRIX Modules		
BAND	Libraries	libsundials_sunmatrixband.LIB
	Headers	sunmatrix/sunmatrix_band.h
CUSPARSE	Libraries	libsundials_sunmatrixcusparse.LIB
	Headers	sunmatrix/sunmatrix_cusparse.h
DENSE	Libraries	libsundials_sunmatrixdense.LIB
	Headers	sunmatrix/sunmatrix_dense.h
MAGMADENSE	Libraries	libsundials_sunmatrixmagmadense.LIB
	Headers	sunmatrix/sunmatrix_magmadense.h
ONEMKLDENSE	Libraries	libsundials_sunmatrixonemkldense.LIB

continues on next page

Table 10.1 – continued from previous page

	Headers	sunmatrix/sunmatrix_onemkldense.h
SPARSE	Libraries	libsundials_sunmatrixsparse.LIB
	Headers	sunmatrix/sunmatrix_sparse.h
SLUNRLOC	Libraries	libsundials_sunmatrixslunrloc.LIB
	Headers	sunmatrix/sunmatrix_slunrloc.h
SUNLINSOL Modules		
BAND	Libraries	libsundials_sunlinsolband.LIB
	Headers	sunlinsol/sunlinsol_band.h
CUSOLVERSP_BATCHQR	Libraries	libsundials_sunlinsolcusolversp.LIB
	Headers	sunlinsol/sunlinsol_cusolversp_batchqr.h
DENSE	Libraries	libsundials_sunlinsoldense.LIB
	Headers	sunlinsol/sunlinsol_dense.h
KLU	Libraries	libsundials_sunlinsolklu.LIB
	Headers	sunlinsol/sunlinsol_klu.h
LAPACKBAND	Libraries	libsundials_sunlinsollapackband.LIB
	Headers	sunlinsol/sunlinsol_lapackband.h
LAPACKDENSE	Libraries	libsundials_sunlinsollapackdense.LIB
	Headers	sunlinsol/sunlinsol_lapackdense.h
MAGMADENSE	Libraries	libsundials_sunlinsolmagmadense.LIB
	Headers	sunlinsol/sunlinsol_magmadense.h
ONEMKLDENSE	Libraries	libsundials_sunlinsolonemkldense.LIB
	Headers	sunlinsol/sunlinsol_onemkldense.h
PCG	Libraries	libsundials_sunlinsolpcg.LIB
	Headers	sunlinsol/sunlinsol_pcg.h
SPBCGS	Libraries	libsundials_sunlinsolspbcgs.LIB
	Headers	sunlinsol/sunlinsol_spbcgs.h
SPFGMR	Libraries	libsundials_sunlinsolspfgmr.LIB
	Headers	sunlinsol/sunlinsol_spfgmr.h
SPGMR	Libraries	libsundials_sunlinsolspgmr.LIB
	Headers	sunlinsol/sunlinsol_spgmr.h
SPTFQMR	Libraries	libsundials_sunlinsolsptfqmr.LIB
	Headers	sunlinsol/sunlinsol_sptfqmr.h
SUPERLUDIST	Libraries	libsundials_sunlinsolsuperludist.LIB
	Headers	sunlinsol/sunlinsol_superludist.h
SUPERLUMT	Libraries	libsundials_sunlinsolsuperlumt.LIB
	Headers	sunlinsol/sunlinsol_superlumt.h
SUNNONLINSOL Modules		
NEWTON	Libraries	libsundials_sunnonlinsolnewton.LIB
	Headers	sunnonlinsol/sunnonlinsol_newton.h
FIXEDPOINT	Libraries	libsundials_sunnonlinsolfixedpoint.LIB
	Headers	sunnonlinsol/sunnonlinsol_fixedpoint.h
PETSCSNES	Libraries	libsundials_sunnonlinsolpetscsnes.LIB
	Headers	sunnonlinsol/sunnonlinsol_petscsnes.h
SUNMEMORY Modules		
SYSTEM	Libraries	libsundials_sunmemsys.LIB
	Headers	sunmemory/sunmemory_system.h
CUDA	Libraries	libsundials_sunmemcuda.LIB
	Headers	sunmemory/sunmemory_cuda.h
HIP	Libraries	libsundials_sunmemhip.LIB
	Headers	sunmemory/sunmemory_hip.h
SYCL	Libraries	libsundials_sunmemsycl.LIB

continues on next page

Table 10.1 – continued from previous page

	Headers	sunmemory/sunmemory_sycl.h
SUNDIALS Packages		
CVODE	Libraries	libsundials_cvode.LIB
	Headers	cvode/cvode.h
		cvode/cvode_bandpre.h
		cvode/cvode_bbdpre.h
		cvode/cvode_diag.h
		cvode/cvode_direct.h
		cvode/cvode_impl.h
		cvode/cvode_ls.h
		cvode/cvode_proj.h
		cvode/cvode_spils.h
CVODES	Libraries	libsundials_cvodes.LIB
	Headers	cvodes/cvodes.h
		cvodes/cvodes_bandpre.h
		cvodes/cvodes_bbdpre.h
		cvodes/cvodes_diag.h
		cvodes/cvodes_direct.h
		cvodes/cvodes_impl.h
		cvodes/cvodes_ls.h
		cvodes/cvodes_spils.h
ARKODE	Libraries	libsundials_arkode.LIB
		libsundials_xbraid.LIB
	Headers	arkode/arkode.h
		arkode/arkode_arkstep.h
		arkode/arkode_bandpre.h
		arkode/arkode_bbdpre.h
		arkode/arkode_butcher.h
		arkode/arkode_butcher_dirk.h
		arkode/arkode_butcher_erk.h
		arkode/arkode_erkstep.h
		arkode/arkode_impl.h
		arkode/arkode_ls.h
		arkode/arkode_mristep.h
		arkode/arkode_xbraid.h
IDA	Libraries	libsundials_ida.LIB
	Headers	ida/ida.h
		ida/ida_bbdpre.h
		ida/ida_direct.h
		ida/ida_impl.h
		ida/ida_ls.h
		ida/ida_spils.h
IDAS	Libraries	libsundials_idas.LIB
	Headers	idas/idas.h
		idas/idas_bbdpre.h
		idas/idas_direct.h
		idas/idas_impl.h
		idas/idas_spils.h
KINSOL	Libraries	libsundials_kinsol.LIB
	Headers	kinsol/kinsol.h
		kinsol/kinsol_bbdpre.h
		kinsol/kinsol_direct.h

continues on next page

Table 10.1 – continued from previous page

		kinsol/kinsol_impl.h
		kinsol/kinsol_ls.h
		kinsol/kinsol_spils.h

Chapter 11

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.

11.1 KINSOL input constants

Table 11.1: KINSOL Main Solver Input Constants

Constant Name	Value	Description
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 Newton iteration.
KIN_LINESEARCH	1	Use Newton iteration with linesearch globalization.
KIN_PICARD	2	Use Picard iteration.

Table 11.2: Iterative Linear Solver Constants

Constant Name	Value	Description
SUN_PREC_NONE	0	No preconditioning
SUN_PREC_RIGHT	2	Preconditioning on the right.
SUN_MODIFIED_GS	1	Use modified Gram-Schmidt procedure.
SUN_CLASSICAL_GS	2	Use classical Gram-Schmidt procedure.

Table 11.3: Anderson Acceleration Orthogonalization Method Constants

Constant Name	Value	Description
KIN_ORTH_MGS	0	Use Modified Gram-Schmidt for Anderson acceleration.
KIN_ORTH_ICWY	1	Use Inverse Compact WY Modified Gram-Schmidt for Anderson acceleration.
KIN_ORTH_CGS2	2	Use Classical Gram-Schmidt with Reorthogonalization (CGS-2) for Anderson Acceleration.
KIN_ORTH_DCGS2	3	Use CGS-2 with Delayed Reorthogonalization for Anderson acceleration.

11.2 KINSOL output constants

Table 11.4: KINSOL Main Solver Output Constants

Constant Name	Value	Description
KIN_SUCCESS	0	Successful function return.
KIN_INITIAL_GUESS_OK	1	The initial user-supplied guess already satisfies the stopping criterion.
KIN_STEP_LT_STPTOL	2	The stopping tolerance on scaled step length was satisfied.
KIN_WARNING	99	A non-fatal warning. The solver will continue.
KIN_MEM_NULL	-1	The kin_mem argument was NULL.
KIN_ILL_INPUT	-2	One of the function inputs is illegal.
KIN_NO_MALLOC	-3	The KINSOL memory was not allocated by a call to KINMalloc.
KIN_MEM_FAIL	-4	A memory allocation failed.
KIN_LINESEARCH_NONCONV	-5	The linesearch algorithm was unable to find an iterate sufficiently distinct from the current iterate.
KIN_MAXITER_REACHED	-6	The maximum number of nonlinear iterations has been reached.
KIN_MXNEWT_5X_EXCEEDED	-7	Five consecutive steps have been taken that satisfy a scaled step length test.
KIN_LINESEARCH_BCFAIL	-8	The linesearch algorithm was unable to satisfy the β-condition for nbcfails iterations.
KIN_LINSOLV_NO_RECOVERY	-9	The user-supplied routine preconditioner slve function failed recoverably, but the preconditioner is already current.
KIN_LINIT_FAIL	-10	The linear solver's initialization function failed.
KIN_LSETUP_FAIL	-11	The linear solver's setup function failed in an unrecoverable manner.
KIN_LSOLVE_FAIL	-12	The linear solver's solve function failed in an unrecoverable manner.
KIN_SYSFUNC_FAIL	-13	The system function failed in an unrecoverable manner.
KIN_FIRST_SYSFUNC_ERR	-14	The system function failed with a recoverable error at the first call.
KIN_REPTD_SYSFUNC_ERR	-15	The system function had repeated recoverable errors.

Table 11.5: KINLS Linear Solver Interface Output Constants

Constant Name	Value	Description
KINLS_SUCCESS	0	Successful function return.
KINLS_MEM_NULL	-1	The kin_mem 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 N_Vector 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 SUNLinearSolver module.

Chapter 12

Appendix: SUNDIALS Release History

Date	SUNDIALS	ARKODE	CVODE	CVODES	IDA	IDAS	KINSOL
Dec 2021	6.0.0	5.0.0	6.0.0	6.0.0	6.0.0	5.0.0	6.0.0
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
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

continues on next page

Table 12.1 – continued from previous page

Date	SUNDIALS	ARKODE	CVODE	CVODES	IDA	IDAS	KINSOL
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]	–	–	–	–

1. CVODE written
2. PVODE written
3. CVODE and PVODE combined
4. IDA written
5. KINSOL written

Bibliography

[1] D. G. Anderson. Iterative procedures for nonlinear integral equations. *J. Assoc. Comput. Machinery*, 12:547–560, 1965.

[2] Cody J Balos, David J Gardner, Carol S Woodward, and Daniel R Reynolds. Enabling GPU accelerated computing in the SUNDIALS time integration library. *Parallel Computing*, 108:102836, 2021.

[3] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. Caliper: performance introspection for hpc software stacks. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 550–560. IEEE, 2016.

[4] P. N. Brown. A local convergence theory for combined inexact-Newton/finite difference projection methods. *SIAM J. Numer. Anal.*, 24(2):407–434, 1987.

[5] P. N. Brown and A. C. Hindmarsh. Reduced Storage Matrix Methods in Stiff ODE Systems. *J. Appl. Math. & Comp.*, 31:49–91, 1989.

[6] P. N. Brown and Y. Saad. Hybrid Krylov Methods for Nonlinear Systems of Equations. *SIAM J. Sci. Stat. Comput.*, 11:450–481, 1990.

[7] G. D. Byrne. Pragmatic Experiments with Krylov Methods in the Stiff ODE Setting. In J.R. Cash and I. Gladwell, editors, *Computational Ordinary Differential Equations*, 323–356. Oxford, 1992. Oxford University Press.

[8] A. M. Collier and R. Serban. Example Programs for KINSOL v6.0.0. Technical Report UCRL-SM-208114, LLNL, 2021.

[9] T. A. Davis and P. N. Ekanathan. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Softw.*, 2010.

[10] R. S. Dembo, S. C. Eisenstat, and T. Steihaug. Inexact Newton Methods. *SIAM J. Numer. Anal.*, 19:400–408, 1982.

[11] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.

[12] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM, Philadelphia, 1996.

[13] 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.

[14] S. C. Eisenstat and H. F. Walker. Choosing the Forcing Terms in an Inexact Newton Method. *SIAM J. Sci. Comput.*, 17:16–32, 1996.

[15] H. Fang and Y. Saad. Two classes of secant methods for nonlinear acceleration. *Numer. Linear Algebra Appl.*, 16:197–221, 2009.

[16] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14:470–482, 1993.

[17] 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.

[18] Vicente Hernández, José E Román, and Andrés Tomás. A parallel variant of the gram-schmidt process with reorthogonalization. In *PARCO*, 221–228. 2005.

[19] 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.

[20] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. \mbox SUNDIALS, suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, pages 363–396, 2005.

[21] A. C. Hindmarsh, R. Serban, and D. R. Reynolds. Example Programs for CVODE v6.0.0. Technical Report, LLNL, 2021. UCRL-SM-208110.

[22] Seth R. Johnson, Andrey Prokopenko, and Katherine J. Evans. Automated fortran-c++ bindings for large-scale scientific applications. 2019. URL: http://arxiv.org/abs/1904.02546, arXiv:1904.02546.

[23] C. T. Kelley. *Iterative Methods for Solving Linear and Nonlinear Equations*. SIAM, Philadelphia, 1995.

[24] X. S. Li. An overview of SuperLU: algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.

[25] 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/\protect\unhbox\voidb@x\protect\penalty\@M\xiaoye/SuperLU/. Last update: August 2011.

[26] 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.

[27] 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.

[28] 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.

[29] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14(2):461–469, 1993. doi:http://dx.doi.org/10.1137/0914028.

[30] 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.

[31] Kasia Swirydowicz, Julien Langou, Shreyas Ananthan, Ulrike Yang, and Stephen Thomas. Low synchronization gram-schmidt and gmres algorithms. *Numerical Linear Algebra with Applications*, 02 2020.

[32] 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. doi:10.1016/j.parco.2009.12.005.

[33] H. A. Van Der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Non-symmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 13:631–644, 1992.

[34] H. F. Walker and P. Ni. Anderson acceleration for fixed-point iterations. *SIAM Jour. Num. Anal.*, 49(4):1715–1735, 2011.

[35] N.a. AMD ROCm Documentation. https://rocmdocs.amd.com/en/latest/index.html.

[36] N.a. Intel oneAPI Programming Guide. https://software.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html.

[37] N.a. KLU Sparse Matrix Factorization Library. http://faculty.cse.tamu.edu/davis/suitesparse.html.

[38] N.a. NVIDIA CUDA Programming Guide. https://docs.nvidia.com/cuda/index.html.

[39] N.a. NVIDIA cuSOLVER Programming Guide. https://docs.nvidia.com/cuda/cusolver/index.html.

[40] N.a. NVIDIA cuSPARSE Programming Guide. https://docs.nvidia.com/cuda/cusparse/index.html.

[41] N.a. SuperLU_DIST Parallel Sparse Matrix Factorization Library. http://crd-legacy.lbl.gov/\protect\unhbox\voidb@x\protect\penalty\@M\xiaoye/SuperLU/.

[42] N.a. SuperLU_MT Threaded Sparse Matrix Factorization Library. http://crd-legacy.lbl.gov/\protect\unhbox\voidb@x\protect\penalty\@M\xiaoye/SuperLU/.

Index

L

M

N

O

P

R

S

U

V

X