User Documentation for IDAS v5.0.0

SUNDIALS v6.0.0

Radu Serban[1], Cosmin Petra[1], Alan C. Hindmarsh[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*

December 15, 2021

UCRL-SM-208112

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

CONTRIBUTORS

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

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

Contents

Chapter 1

Introduction

IDAS is part of a software family called SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic equation Solvers [30]. This suite consists of CVODE, ARKODE, KINSOL, and IDAS, and variants of these with sensitivity analysis capabilities, CVODES and IDAS.

IDAS is a general purpose solver for the initial value problem (IVP) for systems of differential-algebraic equations (DAEs). The name IDAS stands for Implicit Differential-Algebraic solver with Sensitivity capabilities. IDAS is an extension of the IDA solver within SUNDIALS, itself based on on DASPK [8, 9], but is written in ANSI-standard C rather than Fortran77. Its most notable features are that, (1) in the solution of the underlying nonlinear system at each time step, it offers a choice of Newton/direct methods and a choice of Inexact Newton/Krylov (iterative) methods; and (2) it is written in a *data-independent* manner in that it acts on generic vectors and matrices without any assumptions on the underlying organization of the data. Thus IDAS shares significant modules previously written within CASC at LLNL to support the ordinary differential equation (ODE) solvers CVODE [16, 31] and PVODE [12, 13], and also the nonlinear system solver KINSOL [17].

At present, IDAS may utilize a variety of Krylov methods provided in SUNDIALS that can be used in conjuction with Newton iteration: these include the GMRES (Generalized Minimal RESidual) [45], FGMRES (Flexible Generalized Minimum RESidual) [44], Bi-CGStab (Bi-Conjugate Gradient Stabilized) [47], TFQMR (Transpose-Free Quasi-Minimal Residual) [25], and PCG (Preconditioned Conjugate Gradient) [27] 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 DAE 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-CGFStab 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.

IDAS is written with a functionality that is a superset of that of IDA. Sensitivity analysis capabilities, both forward and adjoint, have been added to the main integrator. Enabling forward sensitivity computations in IDAS will result in the code integrating the so-called *sensitivity equations* simultaneously with the original IVP, yielding both the solution and its sensitivity with respect to parameters in the model. Adjoint sensitivity analysis, most useful when the gradients of relatively few functionals of the solution with respect to many parameters are sought, involves integration of the original IVP forward in time followed by the integration of the so-called *adjoint equations* backward in time. IDAS provides the infrastructure needed to integrate any final-condition ODE dependent on the solution of the original IVP (in particular the adjoint system).

1.1 Changes from previous versions

1.1.1 Changes in v5.0.0

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()*

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
IDASpilsSetLinearSolver	IDASetLinearSolver
IDASpilsSetPreconditioner	IDASetPreconditioner
IDASpilsSetJacTimes	IDASetJacTimes
IDASpilsSetEpsLin	IDASetEpsLin
IDASpilsSetIncrementFactor	IDASetIncrementFactor
IDASpilsGetWorkSpace	IDAGetLinWorkSpace
IDASpilsGetNumPrecEvals	IDAGetNumPrecEvals
IDASpilsGetNumPrecSolves	IDAGetNumPrecSolves
IDASpilsGetNumLinIters	IDAGetNumLinIters
IDASpilsGetNumConvFails	IDAGetNumLinConvFails
IDASpilsGetNumJTSetupEvals	IDAGetNumJTSetupEvals
IDASpilsGetNumJtimesEvals	IDAGetNumJtimesEvals
IDASpilsGetNumResEvals	IDAGetNumLinResEvals
IDASpilsGetLastFlag	IDAGetLastLinFlag
IDASpilsGetReturnFlagName	IDAGetLinReturnFlagName
IDASpilsSetLinearSolverB	IDASetLinearSolverB
IDASpilsSetEpsLinB	IDASetEpsLinB
IDASpilsSetIncrementFactorB	IDASetIncrementFactorB
IDASpilsSetPreconditionerB	IDASetPreconditionerB
IDASpilsSetPreconditionerBS	IDASetPreconditionerBS
IDASpilsSetJacTimesB	IDASetJacTimesB
IDASpilsSetJacTimesBS	IDASetJacTimesBS
IDADlsSetLinearSolver	IDASetLinearSolver

continues on next page

Table 1.1 – continued from previous page

Deprecated Name	New Name
IDADlsSetJacFn	IDASetJacFn
IDADlsGetWorkSpace	IDAGetLinWorkSpace
IDADlsGetNumJacEvals	IDAGetNumJacEvals
IDADlsGetNumResEvals	IDAGetNumLinResEvals
IDADlsGetLastFlag	IDAGetLastLinFlag
IDADlsGetReturnFlagName	IDAGetLinReturnFlagName
IDADlsSetLinearSolverB	IDASetLinearSolverB
IDADlsSetJacFnB	IDASetJacFnB
IDADlsSetJacFnBS	IDASetJacFnBS
DenseGETRF	SUNDlsMat_DenseGETRF
DenseGETRS	SUNDlsMat_DenseGETRS
denseGETRF	SUNDlsMat_denseGETRF
denseGETRS	SUNDlsMat_denseGETRS
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

continues on next page

Table 1.1 – continued from previous page

Deprecated Name	New Name
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.

1.1.2 Changes in v4.8.0

The *RAJA N_Vector* implementation has been updated to support the SYCL backend in addition to the CUDA and HIP backends. 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.

IDAS now supports a new "matrix-embedded" `SUNLinearSolver` type. This type supports user-supplied `SUNLinearSolver` implementations that set up and solve 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.

Added the function *IDASetNlsResFn()* to supply an alternative residual side function for use within nonlinear system function evaluations.

The installed `SUNDIALSConfig.cmake` file now supports the `COMPONENTS` option to `find_package`.

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.

1.1.3 Changes in v4.7.0

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.

1.1.4 Changes in v4.6.1

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.

1.1.5 Changes in v4.6.0

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 *NVECTOR_RAJA* 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.

1.1.6 Changes in v4.5.0

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.

1.1.7 Changes in v4.4.0

Added the function *IDASetLSNormFactor()* to specify the factor for converting between integrator tolerances (WRMS norm) and linear solver tolerances (L2 norm) i.e., `tol_L2 = nrmfac * tol_WRMS`.

Added a new function *IDAGetNonlinearSystemData()* which advanced users might find useful if providing a custom *SUNNonlinSolSysFn*.

This change may cause an error in existing user code. The *IDASolveF()* function for forward integration with checkpointing is now subject to a restriction on the number of time steps allowed to reach the output time. This is the same restriction applied to the *IDASolve()* function. The default maximum number of steps is 500, but this may be

changed using the *IDASetMaxNumSteps()* function. This change fixes a bug that could cause an infinite loop in the *IDASolveF()* function.

The expected behavior of *SUNNonlinSolGetNumIters()* and *SUNNonlinSolGetNumConvFails()* in the SUNNonlinearSolver API have been updated to specify that they should return the number of nonlinear solver iterations and convergence failures in the most recent solve respectively rather than the cumulative number of iterations and failures across all solves respectively. The API documentation and SUNDIALS provided SUNNonlinearSolver implementations have been updated accordingly. As before, the cumulative number of nonlinear iterations may be retreived by calling *IDAGetNumNonlinSolvIters()*, the cumulative number of failures with *IDAGetNumNonlinSolvConvFails()*, or both with *IDAGetNonlinSolvStats()*.

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 §4.6 and §10 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.

1.1.8 Changes in v4.3.0

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 a new function *IDAGetNonlinearSystemData()* which advanced users might find useful if providing a custom *SUNNonlinSolSysFn*.

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

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

Added the optional functions *IDASetJacTimesResFn()* and *IDASetJacTimesResFnB()* to specify an alternative residual function for computing Jacobian-vector products with the internal difference quotient approximation.

1.1.9 Changes in v4.2.0

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.

The function *IDASetLinearSolutionScaling()* and IDASetLinearSolutionScalingB was added to enable or disable the scaling applied to linear system solutions with matrix-based linear solvers to account for a lagged value of α in the linear system matrix $J = \frac{\partial F}{\partial y} + \alpha \frac{\partial F}{\partial \dot{y}}$. Scaling is enabled by default when using a matrix-based linear solver.

1.1.10 Changes in v4.1.0

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, FSUNDIALSFileOpen() and FSUNDIALSFileClose() for creating/destroying file pointers that are useful when using the Fortran 2003 interfaces.

1.1.11 Changes in v4.0.0

1.1.11.1 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.

1.1.11.2 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()* 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 for more details.
- Added new Fortran 2003 interfaces for most `N_Vector` modules. See §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.

1.1.11.3 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()* 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 §7.1 for more details.
- The generic `SUNMatrix` API now defines error codes to be returned by `SUNMatrix` operations. Operations which return an integer flag indicating success/failure may return different values than previously.

- 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 §7 for more details on how to use the interfaces.

1.1.11.4 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 for more details.
- Added new Fortran 2003 interfaces for most SUNLinearSolver modules. See §8 for more details on how to use the interfaces.

1.1.11.5 SUNNonlinearSolver module changes

- A new function was added to aid in creating custom SUNNonlinearSolver objects. The constructor *SUNNonlinSolNewEmpty()* allocates an "empty" generic SUNNonlinearSolver 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 SUNNonlinearSolver API by ensuring only required operations need to be set. See §9.1.7 for more details.
- To facilitate the use of user supplied nonlinear solver convergence test functions the *SUNNonlinSolSetConvTestFn* function in the SUNNonlinearSolver API has been updated to take a void* data pointer as input. The supplied data pointer will be passed to the nonlinear solver convergence test function on each call.
- The inputs values passed to the first two inputs of the *SUNNonlinSolSolve()* function in the SUNNonlinearSolver have been changed to be the predicted state and the initial guess for the correction to that state. Additionally, the definitions of *SUNNonlinSolLSetupFn* and *SUNNonlinSolLSolveFn* in the SUNNonlinearSolver API have been updated to remove unused input parameters. For more information see §9.
- Added a new SUNNonlinearSolver implementation, *SUNNONLINSOL_PETSC*, which interfaces to the PETSc SNES nonlinear solver API. See §9.5 for more details.
- Added new Fortran 2003 interfaces for most SUNNonlinearSolver modules. See §9 for more details on how to use the interfaces.

1.1.11.6 IDAS changes

- A bug was fixed in the IDAS linear solver interface where an incorrect Jacobian-vector product increment was used with iterative solvers other than *SUNLINSOL_SPGMR* and *SUNLINSOL_SPFGMR*.
- Fixed a memeory leak in FIDA when not using the default nonlinear solver.
- Fixed a bug where the *IDASolveF()* function would not return a root in `IDA_NORMAL_STEP` mode if the root occurred after the desired output time.
- Fixed a bug where the *IDASolveF()* function would return the wrong flag under certrain cirumstances.
- Fixed a bug in *IDAQuadReInitB()* where an incorrect memory structure was passed to *IDAQuadReInit()*.
- Removed extraneous calls to *N_VMin()* for simulations where the scalar valued absolute tolerance, or all entries of the vector-valued absolute tolerance array, are strictly positive. In this scenario, IDAS will remove at least one global reduction per time step.
- The IDALS 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 the new functions, *IDAGetCurrentCj()*, *IDAGetCurrentY()*, *IDAGetCurrentYp()*, *IDAComputeY()*, and *IDAComputeYp()* which may be useful to users who choose to provide their own nonlinear solver implementations.
- Added a Fortran 2003 interface to IDAS. See §4.5 for more details.

1.1.12 Changes in v3.1.0

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.

A bug was fixed where a nonlinear solver object could be freed twice in some use cases.

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 `idas_impl.h` is no longer installed. This means users who are directly manipulating the `IDAMem` structure will need to update their code to use IDAS's public API.

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

1.1.13 Changes in v3.0.2

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 IDAS library, `libsundials_idas`.

1.1.14 Changes in v3.0.1

No changes were made in this release.

1.1.15 Changes in v3.0.0

IDA's previous direct and iterative linear solver interfaces, IDADLS and IDASPILS, have been merged into a single unified linear solver interface, IDALS, to support any valid `SUNLinearSolver` module. This includes the "DIRECT" and "ITERATIVE" types as well as the new "MATRIX_ITERATIVE" type. Details regarding how IDALS utilizes linear solvers of each type as well as discussion regarding intended use cases for user-supplied `SUNLinearSolver` implementations are included in §8. All IDAS example programs and the standalone linear solver examples have been updated to use the unified linear solver interface.

The unified interface for the new IDALS module is very similar to the previous IDADLS and IDASPILS 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 IDAS 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.

SUNDIALS integrators have been updated to utilize generic nonlinear solver modules defined through the `SUNNonlinearSolver` API. This API will ease the addition of new nonlinear solver options and allow for external or user-supplied nonlinear solvers. The `SUNNonlinearSolver` API and SUNDIALS provided modules are described in §9 and follow the same object oriented design and implementation used by the `N_Vector`, `SUNMatrix`, and `SUNLinearSolver` modules. Currently two `SUNNonlinearSolver` implementations are provided, *SUNNONLINSOL_NEWTON* and *SUNNONLINSOL_FIXEDPOINT*. These replicate the previous integrator specific implementations of a Newton iteration and a fixed-point iteration (previously referred to as a functional iteration), respectively. Note the *SUNNONLINSOL_FIXEDPOINT* module can optionally utilize Anderson's method to accelerate convergence. Example programs using each of these nonlinear solver modules in a standalone manner have been added and all IDAS example programs have been updated to use generic `SUNNonlinearSolver` modules.

By default IDAS uses the *SUNNONLINSOL_NEWTON* module. Since IDAS previously only used an internal implementation of a Newton iteration no changes are required to user programs and functions for setting the nonlinear solver options (e.g., *IDASetMaxNonlinIters()*) or getting nonlinear solver statistics (e.g., *IDAGetNumNonlinSolvIters()*) remain unchanged and internally call generic `SUNNonlinearSolver` functions as needed. While SUNDIALS includes a fixed-point nonlinear solver module, it is not currently supported in IDAS. For details on attaching a user-supplied nonlinear solver to IDAS see §5. Additionally, the example program `idaRoberts_dns.c` explicitly creates an attaches a *SUNNONLINSOL_NEWTON* object to demonstrate the process of creating and attaching a nonlinear solver module (note this is not necessary in general as IDAS uses the *SUNNONLINSOL_NEWTON* module by default).

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 §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.

1.1.16 Changes in v2.2.1

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`. Note *CMAKE_INSTALL_LIBDIR* is automatically set, but is available as a CMake option that can be modified.

1.1.17 Changes in v2.2.0

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.

1.1.18 Changes in v2.1.2

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 *SUNMatScaleAdd()* and *SUNMatScaleAddI()* 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`.

1.1.19 Changes in v2.1.1

The changes in this minor release include the following:

- Fixed a potential memory leak in the *SUNLINSOL_SPGMR* and *SUNLINSOL_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.
- Added missing `#include <stdio.h>` in `N_Vector` and `SUNMatrix` header files.

- Added missing prototype for `IDASpilsGetNumJTSetupEvals()`.
- 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` module (e.g., iterative linear solvers).

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

1.1.20 Changes in v2.1.0

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.

1.1.21 Changes in v2.0.0

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 to ease 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` and `SUNMatrix` modules. The exception is `CVDIAG`, a diagonal approximate Jacobian solver available to CVODE and CVODES.
- Converted all SUNDIALS example problems and files to utilize the new generic `SUNMatrix` and `SUNLinearSolver` objects, along with updated Dls and Spils linear solver interfaces.
- Added Spils interface routines to ARKODE, CVODE, CVODES, IDAS, 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 or residual function evaluation on the device. In addition, these vectors assume the problem fits on one GPU. For further information about RAJA, users are referred to the web site, https://software.llnl.gov/RAJA/. These additions are accompanied by updates 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 add a missing prototype for *IDASetMaxBacksIC()* in `idas.h`.

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

1.1.22 Changes in v1.3.0

Two additional `N_Vector` implementations were added – one for Hypre (parallel) ParVector 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.

An optional input function was added to set a maximum number of linesearch backtracks in the initial condition calculation. Also, corrections were made to three Fortran interface functions.

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 bug in for-loop indices was fixed in `IDAAckpntAllocVectors()`. A bug was fixed in the interpolation functions used in solving backward problems.

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.

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.

New examples were added for use of the OpenMP vector.

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

1.1.23 Changes in v1.2.0

Two major additions were made to the linear system solvers that are available for use with the IDAS 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 IDAS.

Otherwise, only relatively minor modifications were made to IDAS:

In `IDARootfind()`, a minor bug was corrected, where the input array `rootdir` was ignored, and a line was added to break out of root-search loop if the initial interval size is below the tolerance `ttol`.

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

An option was added in the case of Adjoint Sensitivity Analysis with dense or banded Jacobian: With a call to `IDADlsSetDenseJacFnBS` or `IDADlsSetBandJacFnBS`, the user can specify a user-supplied Jacobian function of type `IDADls***JacFnBS`, for the case where the backward problem depends on the forward sensitivities.

A minor bug was fixed regarding the testing of the input `tstop` on the first call to *IDASolve()*.

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 FIDA optional input routines `FIDASETIIN`, `FIDASETRIN`, and `FIDASETVIN`, the optional fourth argument `key_length` was removed, with hardcoded key string lengths passed to all `strncmp` tests.

In all FIDA 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.

1.1.24 Changes in v1.1.0

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` and `NewLintArray`, for `int` and `long int` arrays, respectively.

Errors in the logic for the integration of backward problems were identified and fixed. A large number of minor errors have been fixed. Among these are the following: A missing vector pointer setting was added in `IDASensLineSrch()`. In `IDACompleteStep()`, conditionals around lines loading a new column of three auxiliary divided difference arrays, for a possible order increase, were fixed. After the solver memory is created, it is set to zero before being filled. 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. A memory leak was fixed in two of the `IDASp***Free` functions. In the rootfinding functions `IDARcheck1` and `IDARcheck2`, when an exact zero is found, the array `glo` of `g` values at the left endpoint is adjusted, instead of shifting the `t` location `tlo` slightly. 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 Reading this User Guide

The structure of this document is as follows:

- In Chapter §2, we give short descriptions of the numerical methods implemented by IDAS for the solution of initial value problems for systems of DAEs, along with short descriptions of preconditioning (§2.2) and rootfinding (§2.3).
- The following chapter describes the structure of the SUNDIALS suite of solvers (§3) and the software organization of the IDAS solver (§3.1).
- Chapter §5.1 is the main usage document for IDAS for simulation applications. It includes a complete description of the user interface for the integration of DAE initial value problems. Readers that are not interested in using IDAS for sensitivity analysis can then skip the next two chapters.
- Chapter §5.4 describes the usage of IDAS for forward sensitivity analysis as an extension of its IVP integration capabilities. We begin with a skeleton of the user main program, with emphasis on the steps that are required in addition to those already described in Chapter §5.1. Following that we provide detailed descriptions of the user-callable interface routines specific to forward sensitivity analysis and of the additonal optional user-defined routines.
- Chapter §5.5 describes the usage of IDAS for adjoint sensitivity analysis. We begin by describing the IDAS checkpointing implementation for interpolation of the original IVP solution during integration of the adjoint system backward in time, and with an overview of a user's main program. Following that we provide complete descriptions of the user-callable interface routines for adjoint sensitivity analysis as well as descriptions of the required additional user-defined routines.
- Chapter §6 gives a brief overview of the generic N_Vector module shared among the various components of SUNDIALS, as well as details on the 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.
- Chapter §9 describes the SUNNonlinearSolver API and nonlinear solver implementations shared among the various components of SUNDIALS.
- Finally, in the appendices, we provide detailed instructions for the installation of IDAS, within the structure of SUNDIALS (Appendix §11), as well as a list of all the constants used for input to and output from IDAS functions (Appendix §12).

1.3 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.

1.3.1 BSD 3-Clause License

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

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

1.3.2 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.

Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or Lawrence Livermore National Security, LLC.

The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

1.3.3 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)

Chapter 2

Mathematical Considerations

IDAS solves the initial-value problem (IVP) for a DAE system of the general form

$$F(t, y, \dot{y}) = 0\,, \quad y(t_0) = y_0\,, \quad \dot{y}(t_0) = \dot{y}_0 \tag{2.1}$$

where y, $\dot{y}$, and F are vectors in $\mathbf{R}^N$, t is the independent variable, $\dot{y} = \mathrm{d}y/\mathrm{d}t$, and initial values y_0, $\dot{y}_0$ are given. (Often t is time, but it certainly need not be.)

Additionally, if (2.1) depends on some parameters $p \in \mathbf{R}^{N_p}$, i.e.

$$\begin{aligned} &F(t, y, \dot{y}, p) = 0 \\ &y(t_0) = y_0(p)\,,\ \dot{y}(t_0) = \dot{y}_0(p)\,, \end{aligned} \tag{2.2}$$

IDAS can also compute first order derivative information, performing either *forward sensitivity analysis* or *adjoint sensitivity analysis*. In the first case, IDAS computes the sensitivities of the solution with respect to the parameters p, while in the second case, IDAS computes the gradient of a *derived function* with respect to the parameters p.

2.1 IVP solution

Prior to integrating a DAE initial-value problem, an important requirement is that the pair of vectors y_0 and $\dot{y}_0$ are both initialized to satisfy the DAE residual $F(t_0, y_0, \dot{y}_0) = 0$. For a class of problems that includes so-called semi-explicit index-one systems, IDAS provides a routine that computes consistent initial conditions from a user's initial guess [9]. For this, the user must identify sub-vectors of y (not necessarily contiguous), denoted y_d and y_a, which are its differential and algebraic parts, respectively, such that F depends on $\dot{y}_d$ but not on any components of $\dot{y}_a$. The assumption that the system is "index one" means that for a given t and y_d, the system $F(t, y, \dot{y}) = 0$ defines y_a uniquely. In this case, a solver within IDAS computes y_a and $\dot{y}_d$ at $t = t_0$, given y_d and an initial guess for y_a. A second available option with this solver also computes all of $y(t_0)$ given $\dot{y}(t_0)$; this is intended mainly for quasi-steady-state problems, where $\dot{y}(t_0) = 0$ is given. In both cases, IDAS solves the system $F(t_0, y_0, \dot{y}_0) = 0$ for the unknown components of y_0 and $\dot{y}_0$, using Newton iteration augmented with a line search global strategy. In doing this, it makes use of the existing machinery that is to be used for solving the linear systems during the integration, in combination with certain tricks involving the step size (which is set artificially for this calculation). For problems that do not fall into either of these categories, the user is responsible for passing consistent values, or risks failure in the numerical integration.

The integration method used in IDAS is the variable-order, variable-coefficient BDF (Backward Differentiation Formula), in fixed-leading-coefficient form [4]. The method order ranges from 1 to 5, with the BDF of order q given by the multistep formula

$$\sum_{i=0}^{q} \alpha_{n,i} y_{n-i} = h_n \dot{y}_n\,, \tag{2.3}$$

where y_n and $\dot{y}_n$ are the computed approximations to $y(t_n)$ and $\dot{y}(t_n)$, respectively, and the step size is $h_n = t_n - t_{n-1}$. The coefficients $\alpha_{n,i}$ are uniquely determined by the order q, and the history of the step sizes. The application of the BDF (2.3) to the DAE system (2.1) results in a nonlinear algebraic system to be solved at each step:

$$G(y_n) \equiv F\left(t_n,\, y_n,\, h_n^{-1} \sum_{i=0}^{q} \alpha_{n,i} y_{n-i}\right) = 0\,. \tag{2.4}$$

By default IDAS solves (2.4) with a Newton iteration but IDAS also allows for user-defined nonlinear solvers (see Chapter §9). Each Newton iteration requires the solution of a linear system of the form

$$J[y_{n(m+1)} - y_{n(m)}] = -G(y_{n(m)})\,, \tag{2.5}$$

where $y_{n(m)}$ is the m-th approximation to y_n. Here J is some approximation to the system Jacobian

$$J = \frac{\partial G}{\partial y} = \frac{\partial F}{\partial y} + \alpha \frac{\partial F}{\partial \dot{y}}\,, \tag{2.6}$$

where $\alpha = \alpha_{n,0}/h_n$. The scalar α changes whenever the step size or method order changes.

For the solution of the linear systems within the Newton iteration, IDAS provides several choices, including the option of a user-supplied linear solver (see Chapter §8). The linear solvers 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, including an internal implementation, an interface to BLAS/LAPACK, an interface to MAGMA [46] and an interface to the oneMKL library [50],
- band direct solvers, including an internal implementation or an interface to BLAS/LAPACK,
- sparse direct solver interfaces to various libraries, including KLU [18, 51], SuperLU_MT [20, 38, 56], SuperLU_Dist [26, 39, 40, 55], and cuSPARSE [54],
- 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.

For large stiff systems, where direct methods are not feasible, the combination of a BDF integrator and a preconditioned Krylov method yields a powerful tool because it combines established methods for stiff integration, nonlinear iteration, and Krylov (linear) iteration with a problem-specific treatment of the dominant source of stiffness, in the form of the user-supplied preconditioner matrix [7]. For the *spils* linear solvers with IDAS, preconditioning is allowed only on the left (see §2.2). Note that the dense, band, and sparse direct linear solvers can only be used with serial and threaded vector representations.

In the process of controlling errors at various levels, IDAS uses a weighted root-mean-square norm, denoted $\|\cdot\|_{\text{WRMS}}$, for all error-like quantities. The multiplicative weights used are based on the current solution and on the relative and absolute tolerances input by the user, namely

$$W_i = \frac{1}{\text{rtol} \cdot |y_i| + \text{atol}_i}\,. \tag{2.7}$$

Because $1/W_i$ represents a tolerance in the component y_i, a vector whose norm is 1 is regarded as "small." For brevity, we will usually drop the subscript WRMS on norms in what follows.

In the case of a matrix-based linear solver, the default Newton iteration is a Modified Newton iteration, in that the Jacobian J is fixed (and usually out of date) throughout the nonlinear iterations, with a coefficient $\bar{\alpha}$ in place of α in J.

However, in the case that a matrix-free iterative linear solver is used, the default Newton iteration is an Inexact Newton iteration, in which J is applied in a matrix-free manner, with matrix-vector products Jv obtained by either difference quotients or a user-supplied routine. In this case, the linear residual $J\Delta y + G$ is nonzero but controlled. With the default Newton iteration, the matrix J and preconditioner matrix P are updated as infrequently as possible to balance the high costs of matrix operations against other costs. Specifically, this matrix update occurs when:

- starting the problem,
- the value $\bar{\alpha}$ at the last update is such that $\alpha/\bar{\alpha} < 3/5$ or $\alpha/\bar{\alpha} > 5/3$, or
- a non-fatal convergence failure occurred with an out-of-date J or P.

The above strategy balances the high cost of frequent matrix evaluations and preprocessing with the slow convergence due to infrequent updates. To reduce storage costs on an update, Jacobian information is always reevaluated from scratch.

The default stopping test for nonlinear solver iterations in IDAS ensures that the iteration error $y_n - y_{n(m)}$ is small relative to y itself. For this, we estimate the linear convergence rate at all iterations $m > 1$ as

$$R = \left(\frac{\delta_m}{\delta_1}\right)^{\frac{1}{m-1}} ,$$

where the $\delta_m = y_{n(m)} - y_{n(m-1)}$ is the correction at iteration $m = 1, 2, \ldots$. The nonlinear solver iteration is halted if $R > 0.9$. The convergence test at the m-th iteration is then

$$S\|\delta_m\| < 0.33 , \tag{2.8}$$

where $S = R/(R-1)$ whenever $m > 1$ and $R \leq 0.9$. The user has the option of changing the constant in the convergence test from its default value of 0.33. The quantity S is set to $S = 20$ initially and whenever J or P is updated, and it is reset to $S = 100$ on a step with $\alpha \neq \bar{\alpha}$. Note that at $m = 1$, the convergence test (2.8) uses an old value for S. Therefore, at the first nonlinear solver iteration, we make an additional test and stop the iteration if $\|\delta_1\| < 0.33 \cdot 10^{-4}$ (since such a δ_1 is probably just noise and therefore not appropriate for use in evaluating R). We allow only a small number (default value 4) of nonlinear iterations. If convergence fails with J or P current, we are forced to reduce the step size h_n, and we replace h_n by $h_n/4$. The integration is halted after a preset number (default value 10) of convergence failures. Both the maximum number of allowable nonlinear iterations and the maximum number of nonlinear convergence failures can be changed by the user from their default values.

When an iterative method is used to solve the linear system, to minimize the effect of linear iteration errors on the nonlinear and local integration error controls, we require the preconditioned linear residual to be small relative to the allowed error in the nonlinear iteration, i.e., $\|P^{-1}(Jx + G)\| < 0.05 \cdot 0.33$. The safety factor 0.05 can be changed by the user.

When the Jacobian is stored using either the *SUNMATRIX_DENSE* or *SUNMATRIX_BAND* matrix objects, the Jacobian J defined in (2.6) can be either supplied by the user or IDAS can compute J internally by difference quotients. In the latter case, we use the approximation

$$\begin{aligned} J_{ij} &= [F_i(t, y + \sigma_j e_j, \dot{y} + \alpha\sigma_j e_j) - F_i(t, y, \dot{y})]/\sigma_j , \text{ with} \\ \sigma_j &= \sqrt{U} \max\{|y_j|, |h\dot{y}_j|, 1/W_j\} \operatorname{sign}(h\dot{y}_j) , \end{aligned}$$

where U is the unit roundoff, h is the current step size, and W_j is the error weight for the component y_j defined by (2.7). We note that with sparse and user-supplied matrix objects, the Jacobian *must* be supplied by a user routine.

In the case of an iterative linear solver, if a routine for Jv is not supplied, such products are approximated by

$$Jv = [F(t, y + \sigma v, \dot{y} + \alpha\sigma v) - F(t, y, \dot{y})]/\sigma ,$$

where the increment $\sigma = 1/\|v\|$. As an option, the user can specify a constant factor that is inserted into this expression for σ.

During the course of integrating the system, IDAS computes an estimate of the local truncation error, LTE, at the n-th time step, and requires this to satisfy the inequality

$$\|\text{LTE}\|_{\text{WRMS}} \leq 1\,.$$

Asymptotically, LTE varies as h^{q+1} at step size h and order q, as does the predictor-corrector difference $\Delta_n \equiv y_n - y_{n(0)}$. Thus there is a constant C such that

$$\text{LTE} = C\Delta_n + O(h^{q+2})\,,$$

and so the norm of LTE is estimated as $|C|\cdot\|\Delta_n\|$. In addition, IDAS requires that the error in the associated polynomial interpolant over the current step be bounded by 1 in norm. The leading term of the norm of this error is bounded by $\bar{C}\|\Delta_n\|$ for another constant $\bar{C}$. Thus the local error test in IDAS is

$$\max\{|C|, \bar{C}\}\|\Delta_n\| \leq 1\,. \tag{2.9}$$

A user option is available by which the algebraic components of the error vector are omitted from the test (2.9), if these have been so identified.

In IDAS, the local error test is tightly coupled with the logic for selecting the step size and order. First, there is an initial phase that is treated specially; for the first few steps, the step size is doubled and the order raised (from its initial value of 1) on every step, until (a) the local error test (2.9) fails, (b) the order is reduced (by the rules given below), or (c) the order reaches 5 (the maximum). For step and order selection on the general step, IDAS uses a different set of local error estimates, based on the asymptotic behavior of the local error in the case of fixed step sizes. At each of the orders q' equal to q, $q-1$ (if $q > 1$), $q-2$ (if $q > 2$), or $q+1$ (if $q < 5$), there are constants $C(q')$ such that the norm of the local truncation error at order q' satisfies

$$\text{LTE}(q') = C(q')\|\phi(q'+1)\| + O(h^{q'+2})\,,$$

where $\phi(k)$ is a modified divided difference of order k that is retained by IDAS (and behaves asymptotically as h^k). Thus the local truncation errors are estimated as $\text{ELTE}(q') = C(q')\|\phi(q'+1)\|$ to select step sizes. But the choice of order in IDAS is based on the requirement that the scaled derivative norms, $\|h^k y^{(k)}\|$, are monotonically decreasing with k, for k near q. These norms are again estimated using the $\phi(k)$, and in fact

$$\|h^{q'+1}y^{(q'+1)}\| \approx T(q') \equiv (q'+1)\text{ELTE}(q')\,.$$

The step/order selection begins with a test for monotonicity that is made even *before* the local error test is performed. Namely, the order is reset to $q' = q-1$ if (a) $q = 2$ and $T(1) \leq T(2)/2$, or (b) $q > 2$ and $\max\{T(q-1), T(q-2)\} \leq T(q)$; otherwise $q' = q$. Next the local error test (2.9) is performed, and if it fails, the step is redone at order $q \leftarrow q'$ and a new step size h'. The latter is based on the h^{q+1} asymptotic behavior of $\text{ELTE}(q)$, and, with safety factors, is given by

$$\eta = h'/h = 0.9/[2\,\text{ELTE}(q)]^{1/(q+1)}\,.$$

The value of η is adjusted so that $0.25 \leq \eta \leq 0.9$ before setting $h \leftarrow h' = \eta h$. If the local error test fails a second time, IDAS uses $\eta = 0.25$, and on the third and subsequent failures it uses $q = 1$ and $\eta = 0.25$. After 10 failures, IDAS returns with a give-up message.

As soon as the local error test has passed, the step and order for the next step may be adjusted. No such change is made if $q' = q-1$ from the prior test, if $q = 5$, or if q was increased on the previous step. Otherwise, if the last $q+1$ steps were taken at a constant order $q < 5$ and a constant step size, IDAS considers raising the order to $q+1$. The logic is as follows: (a) If $q = 1$, then reset $q = 2$ if $T(2) < T(1)/2$. (b) If $q > 1$ then

- reset $q \leftarrow q-1$ if $T(q-1) \leq \min\{T(q), T(q+1)\}$;
- else reset $q \leftarrow q+1$ if $T(q+1) < T(q)$;
- leave q unchanged otherwise [then $T(q-1) > T(q) \leq T(q+1)$].

In any case, the new step size h' is set much as before:

$$\eta = h'/h = 1/[2\,\mathrm{ELTE}(q)]^{1/(q+1)}\,.$$

The value of η is adjusted such that (a) if $\eta > 2$, η is reset to 2; (b) if $\eta \leq 1$, η is restricted to $0.5 \leq \eta \leq 0.9$; and (c) if $1 < \eta < 2$ we use $\eta = 1$. Finally h is reset to $h' = \eta h$. Thus we do not increase the step size unless it can be doubled. See [4] for details.

IDAS permits the user to impose optional inequality constraints on individual components of the solution vector y. Any of the following four constraints can be imposed: $y_i > 0$, $y_i < 0$, $y_i \geq 0$, or $y_i \leq 0$. The constraint satisfaction is tested after a successful nonlinear system solution. If any constraint fails, we declare a convergence failure of the nonlinear iteration and reduce the step size. Rather than cutting the step size by some arbitrary factor, IDAS estimates a new step size h' using a linear approximation of the components in y that failed the constraint test (including a safety factor of 0.9 to cover the strict inequality case). These additional constraints are also imposed during the calculation of consistent initial conditions. If a step fails to satisfy the constraints repeatedly within a step attempt then the integration is halted and an error is returned. In this case the user may need to employ other strategies as discussed in §5.1.4.2 to satisfy the inequality constraints.

Normally, IDAS takes steps until a user-defined output value $t = t_{\mathrm{out}}$ is overtaken, and then computes $y(t_{\mathrm{out}})$ by interpolation. However, a "one step" mode option is available, where control returns to the calling program after each step. There are also options to force IDAS not to integrate past a given stopping point $t = t_{\mathrm{stop}}$.

2.2 Preconditioning

When using a nonlinear solver that requires the solution of a linear system of the form $J\Delta y = -G$ (e.g., the default Newton iteration), IDAS makes repeated use of a linear solver. If this linear system solve is done with one of the scaled preconditioned iterative linear solvers supplied with SUNDIALS, these solvers are rarely successful if used without preconditioning; it is generally necessary to precondition the system in order to obtain acceptable efficiency. A system $Ax = b$ can be preconditioned on the left, on the right, or on both sides. The Krylov method is then applied to a system with the matrix $P^{-1}A$, or AP^{-1}, or $P_L^{-1}AP_R^{-1}$, instead of A. However, within IDAS, preconditioning is allowed *only* on the left, so that the iterative method is applied to systems $(P^{-1}J)\Delta y = -P^{-1}G$. Left preconditioning is required to make the norm of the linear residual in the nonlinear iteration meaningful; in general, $\|J\Delta y + G\|$ is meaningless, since the weights used in the WRMS-norm correspond to y.

In order to improve the convergence of the Krylov iteration, the preconditioner matrix P should in some sense approximate the system matrix A. Yet at the same time, in order to be cost-effective, the matrix P should be reasonably efficient to evaluate and solve. Finding a good point in this tradeoff between rapid convergence and low cost can be very difficult. Good choices are often problem-dependent (for example, see [7] for an extensive study of preconditioners for reaction-transport systems).

Typical preconditioners used with IDAS are based on approximations to the iteration matrix of the systems involved; in other words, $P \approx \partial F/\partial y + \alpha\,\partial F/\partial \dot{y}$, where α is a scalar inversely proportional to the integration step size h. Because the Krylov iteration occurs within a nonlinear solver iteration and further also within a time integration, and since each of these iterations has its own test for convergence, the preconditioner may use a very crude approximation, as long as it captures the dominant numerical feature(s) of the system. We have found that the combination of a preconditioner with the Newton-Krylov iteration, using even a fairly poor approximation to the Jacobian, can be surprisingly superior to using the same matrix without Krylov acceleration (i.e., a modified Newton iteration), as well as to using the Newton-Krylov method with no preconditioning.

2.3 Rootfinding

The IDAS solver has been augmented to include a rootfinding feature. This means that, while integratnuming the Initial Value Problem (2.1), IDAS can also find the roots of a set of user-defined functions $g_i(t, y, \dot{y})$ that depend on t, the solution vector $y = y(t)$, and its $t-$derivative $\dot{y}(t)$. The number of these root functions is arbitrary, and if more than one g_i is found to have a root in any given interval, the various root locations are found and reported in the order that they occur on the t axis, in the direction of integration.

Generally, this rootfinding feature finds only roots of odd multiplicity, corresponding to changes in sign of $g_i(t, y(t), \dot{y}(t))$, denoted $g_i(t)$ for short. If a user root function has a root of even multiplicity (no sign change), it will probably be missed by IDAS. If such a root is desired, the user should reformulate the root function so that it changes sign at the desired root.

The basic scheme used is to check for sign changes of any $g_i(t)$ over each time step taken, and then (when a sign change is found) to home in on the root (or roots) with a modified secant method [28]. In addition, each time g is computed, IDAS checks to see if $g_i(t) = 0$ exactly, and if so it reports this as a root. However, if an exact zero of any g_i is found at a point t, IDAS computes g at $t + \delta$ for a small increment δ, slightly further in the direction of integration, and if any $g_i(t + \delta) = 0$ also, IDAS stops and reports an error. This way, each time IDAS takes a time step, it is guaranteed that the values of all g_i are nonzero at some past value of t, beyond which a search for roots is to be done.

At any given time in the course of the time-stepping, after suitable checking and adjusting has been done, IDAS has an interval $(t_{lo}, t_{hi}]$ in which roots of the $g_i(t)$ are to be sought, such that t_{hi} is further ahead in the direction of integration, and all $g_i(t_{lo}) \neq 0$. The endpoint t_{hi} is either t_n, the end of the time step last taken, or the next requested output time t_{out} if this comes sooner. The endpoint t_{lo} is either t_{n-1}, or the last output time t_{out} (if this occurred within the last step), or the last root location (if a root was just located within this step), possibly adjusted slightly toward t_n if an exact zero was found. The algorithm checks g at t_{hi} for zeros and for sign changes in (t_{lo}, t_{hi}). If no sign changes are found, then either a root is reported (if some $g_i(t_{hi}) = 0$) or we proceed to the next time interval (starting at t_{hi}). If one or more sign changes were found, then a loop is entered to locate the root to within a rather tight tolerance, given by

$$\tau = 100 * U * (|t_n| + |h|) \quad (U = \text{unit roundoff}) \ .$$

Whenever sign changes are seen in two or more root functions, the one deemed most likely to have its root occur first is the one with the largest value of $|g_i(t_{hi})|/|g_i(t_{hi}) - g_i(t_{lo})|$, corresponding to the closest to t_{lo} of the secant method values. At each pass through the loop, a new value t_{mid} is set, strictly within the search interval, and the values of $g_i(t_{mid})$ are checked. Then either t_{lo} or t_{hi} is reset to t_{mid} according to which subinterval is found to have the sign change. If there is none in (t_{lo}, t_{mid}) but some $g_i(t_{mid}) = 0$, then that root is reported. The loop continues until $|t_{hi} - t_{lo}| < \tau$, and then the reported root location is t_{hi}.

In the loop to locate the root of $g_i(t)$, the formula for t_{mid} is

$$t_{mid} = t_{hi} - (t_{hi} - t_{lo})g_i(t_{hi})/[g_i(t_{hi}) - \alpha g_i(t_{lo})] \ ,$$

where α a weight parameter. On the first two passes through the loop, α is set to 1, making t_{mid} the secant method value. Thereafter, α is reset according to the side of the subinterval (low vs high, i.e. toward t_{lo} vs toward t_{hi}) in which the sign change was found in the previous two passes. If the two sides were opposite, α is set to 1. If the two sides were the same, α is halved (if on the low side) or doubled (if on the high side). The value of t_{mid} is closer to t_{lo} when $\alpha < 1$ and closer to t_{hi} when $\alpha > 1$. If the above value of t_{mid} is within $\tau/2$ of t_{lo} or t_{hi}, it is adjusted inward, such that its fractional distance from the endpoint (relative to the interval size) is between 0.1 and 0.5 (0.5 being the midpoint), and the actual distance from the endpoint is at least $\tau/2$.

2.4 Pure quadrature integration

In many applications, and most notably during the backward integration phase of an adjoint sensitivity analysis run §2.6 it is of interest to compute integral quantities of the form

$$z(t) = \int_{t_0}^{t} q(\tau, y(\tau), \dot{y}(\tau), p)\, d\tau\,. \tag{2.10}$$

The most effective approach to compute $z(t)$ is to extend the original problem with the additional ODEs (obtained by applying Leibnitz's differentiation rule):

$$\dot{z} = q(t, y, \dot{y}, p)\,, \quad z(t_0) = 0\,.$$

Note that this is equivalent to using a quadrature method based on the underlying linear multistep polynomial representation for $y(t)$.

This can be done at the "user level" by simply exposing to IDAS the extended DAE system (2.2) + (2.10). However, in the context of an implicit integration solver, this approach is not desirable since the nonlinear solver module will require the Jacobian (or Jacobian-vector product) of this extended DAE. Moreover, since the additional states, z, do not enter the right-hand side of the ODE (2.10) and therefore the residual of the extended DAE system does not depend on z, it is much more efficient to treat the ODE system (2.10) separately from the original DAE system (2.2) by "taking out" the additional states z from the nonlinear system (2.4) that must be solved in the correction step of the LMM. Instead, "corrected" values z_n are computed explicitly as

$$z_n = \frac{1}{\alpha_{n,0}} \left(h_n q(t_n, y_n, \dot{y}_n, p) - \sum_{i=1}^{q} \alpha_{n,i} z_{n-i} \right)\,,$$

once the new approximation y_n is available.

The quadrature variables z can be optionally included in the error test, in which case corresponding relative and absolute tolerances must be provided.

2.5 Forward sensitivity analysis

Typically, the governing equations of complex, large-scale models depend on various parameters, through the right-hand side vector and/or through the vector of initial conditions, as in (2.2). In addition to numerically solving the DAEs, it may be desirable to determine the sensitivity of the results with respect to the model parameters. Such sensitivity information can be used to estimate which parameters are most influential in affecting the behavior of the simulation or to evaluate optimization gradients (in the setting of dynamic optimization, parameter estimation, optimal control, etc.).

The *solution sensitivity* with respect to the model parameter p_i is defined as the vector $s_i(t) = \partial y(t)/\partial p_i$ and satisfies the following *forward sensitivity equations* (or *sensitivity equations* for short):

$$\begin{aligned} &\frac{\partial F}{\partial y} s_i + \frac{\partial F}{\partial \dot{y}} \dot{s}_i + \frac{\partial F}{\partial p_i} = 0 \\ &s_i(t_0) = \frac{\partial y_0(p)}{\partial p_i}\,,\ \dot{s}_i(t_0) = \frac{\partial \dot{y}_0(p)}{\partial p_i}\,, \end{aligned} \tag{2.11}$$

obtained by applying the chain rule of differentiation to the original DAEs (2.2).

When performing forward sensitivity analysis, IDAS carries out the time integration of the combined system, (2.2) and (2.11), by viewing it as a DAE system of size $N(N_s + 1)$, where N_s is the number of model parameters p_i, with respect to which sensitivities are desired ($N_s \leq N_p$). However, major improvements in efficiency can be made by

taking advantage of the special form of the sensitivity equations as linearizations of the original DAEs. In particular, the original DAE system and all sensitivity systems share the same Jacobian matrix J in (2.6).

The sensitivity equations are solved with the same linear multistep formula that was selected for the original DAEs and the same linear solver is used in the correction phase for both state and sensitivity variables. In addition, IDAS offers the option of including (*full error control*) or excluding (*partial error control*) the sensitivity variables from the local error test.

2.5.1 Forward sensitivity methods

In what follows we briefly describe three methods that have been proposed for the solution of the combined DAE and sensitivity system for the vector $\hat{y} = [y, s_1, \ldots, s_{N_s}]$.

- *Staggered Direct* In this approach [15], the nonlinear system (2.4) is first solved and, once an acceptable numerical solution is obtained, the sensitivity variables at the new step are found by directly solving (2.11) after the BDF discretization is used to eliminate $\dot{s}_i$. Although the system matrix of the above linear system is based on exactly the same information as the matrix J in (2.6), it must be updated and factored at every step of the integration, in contrast to an evaluation of J which is updated only occasionally. For problems with many parameters (relative to the problem size), the staggered direct method can outperform the methods described below [37]. However, the computational cost associated with matrix updates and factorizations makes this method unattractive for problems with many more states than parameters (such as those arising from semidiscretization of PDEs) and is therefore not implemented in IDAS.

- *Simultaneous Corrector* In this method [42], the discretization is applied simultaneously to both the original equations (2.2) and the sensitivity systems (2.11) resulting in an "extended" nonlinear system $\hat{G}(\hat{y}_n) = 0$ where $\hat{y}_n = [y_n, \ldots, s_i, \ldots]$. This combined nonlinear system can be solved using a modified Newton method as in (2.5) by solving the corrector equation

$$\hat{J}[\hat{y}_{n(m+1)} - \hat{y}_{n(m)}] = -\hat{G}(\hat{y}_{n(m)}) \tag{2.12}$$

  at each iteration, where

$$\hat{J} = \begin{bmatrix} J & & & & \\ J_1 & J & & & \\ J_2 & 0 & J & & \\ \vdots & \vdots & \ddots & \ddots & \\ J_{N_s} & 0 & \ldots & 0 & J \end{bmatrix},$$

  J is defined as in (2.6), and $J_i = (\partial/\partial y)\,[F_y s_i + F_{\dot{y}} \dot{s}_i + F_{p_i}]$. It can be shown that 2-step quadratic convergence can be retained by using only the block-diagonal portion of $\hat{J}$ in the corrector equation (2.12). This results in a decoupling that allows the reuse of J without additional matrix factorizations. However, the sum $F_y s_i + F_{\dot{y}} \dot{s}_i + F_{p_i}$ must still be reevaluated at each step of the iterative process (2.12) to update the sensitivity portions of the residual $\hat{G}$.

- *Staggered corrector* In this approach [24], as in the staggered direct method, the nonlinear system (2.4) is solved first using the Newton iteration (2.5). Then, for each sensitivity vector $\xi \equiv s_i$, a separate Newton iteration is used to solve the sensitivity system (2.11):

$$J[\xi_{n(m+1)} - \xi_{n(m)}] = \\ -\left[F_y(t_n, y_n, \dot{y}_n)\xi_{n(m)} + F_{\dot{y}}(t_n, y_n, \dot{y}_n) \cdot h_n^{-1} \left(\alpha_{n,0}\xi_{n(m)} + \sum_{i=1}^{q} \alpha_{n,i}\xi_{n-i}\right) + F_{p_i}(t_n, y_n, \dot{y}_n)\right]. \tag{2.13}$$

  In other words, a modified Newton iteration is used to solve a linear system. In this approach, the matrices $\partial F/\partial y$, $\partial F/\partial \dot{y}$ and vectors $\partial f/\partial p_i$ need be updated only once per integration step, after the state correction phase (2.5) has converged.

IDAS implements both the simultaneous corrector method and the staggered corrector method.

An important observation is that the staggered corrector method, combined with a Krylov linear solver, effectively results in a staggered direct method. Indeed, the Krylov solver requires only the action of the matrix J on a vector, and this can be provided with the current Jacobian information. Therefore, the modified Newton procedure (2.13) will theoretically converge after one iteration.

2.5.2 Selection of the absolute tolerances for sensitivity variables

If the sensitivities are included in the error test, IDAS provides an automated estimation of absolute tolerances for the sensitivity variables based on the absolute tolerance for the corresponding state variable. The relative tolerance for sensitivity variables is set to be the same as for the state variables. The selection of absolute tolerances for the sensitivity variables is based on the observation that the sensitivity vector s_i will have units of $[y]/[p_i]$. With this, the absolute tolerance for the j-th component of the sensitivity vector s_i is set to $\text{atol}_j/|\bar{p}_i|$, where atol_j are the absolute tolerances for the state variables and $\bar{p}$ is a vector of scaling factors that are dimensionally consistent with the model parameters p and give an indication of their order of magnitude. This choice of relative and absolute tolerances is equivalent to requiring that the weighted root-mean-square norm of the sensitivity vector s_i with weights based on s_i be the same as the weighted root-mean-square norm of the vector of scaled sensitivities $\bar{s}_i = |\bar{p}_i|s_i$ with weights based on the state variables (the scaled sensitivities $\bar{s}_i$ being dimensionally consistent with the state variables). However, this choice of tolerances for the s_i may be a poor one, and the user of IDAS can provide different values as an option.

2.5.3 Evaluation of the sensitivity right-hand side

There are several methods for evaluating the residual functions in the sensitivity systems (2.11): analytic evaluation, automatic differentiation, complex-step approximation, and finite differences (or directional derivatives). IDAS provides all the software hooks for implementing interfaces to automatic differentiation (AD) or complex-step approximation; future versions will include a generic interface to AD-generated functions. At the present time, besides the option for analytical sensitivity right-hand sides (user-provided), IDAS can evaluate these quantities using various finite difference-based approximations to evaluate the terms $(\partial F/\partial y)s_i+(\partial F/\partial \dot{y})\dot{s}_i$ and $(\partial f/\partial p_i)$, or using directional derivatives to evaluate $[(\partial F/\partial y)s_i + (\partial F/\partial \dot{y})\dot{s}_i + (\partial f/\partial p_i)]$. As is typical for finite differences, the proper choice of perturbations is a delicate matter. IDAS takes into account several problem-related features: the relative DAE error tolerance rtol, the machine unit roundoff U, the scale factor $\bar{p}_i$, and the weighted root-mean-square norm of the sensitivity vector s_i.

Using central finite differences as an example, the two terms $(\partial F/\partial y)s_i + (\partial F/\partial \dot{y})\dot{s}_i$ and $\partial f/\partial p_i$ in (2.11) can be evaluated either separately:

$$\frac{\partial F}{\partial y}s_i + \frac{\partial F}{\partial y}p\dot{s}_i \approx \frac{F(t, y+\sigma_y s_i, \dot{y}+\sigma_y \dot{s}_i, p) - F(t, y-\sigma_y s_i, \dot{y}-\sigma_y \dot{s}_i, p)}{2\,\sigma_y}\,, \tag{2.14}$$

$$\frac{\partial F}{\partial p_i} \approx \frac{F(t, y, \dot{y}, p+\sigma_i e_i) - F(t, y, \dot{y}, p-\sigma_i e_i)}{2\,\sigma_i}\,, \tag{2.15}$$

$$\sigma_i = |\bar{p}_i|\sqrt{\max(\text{rtol}, U)}\,, \quad \sigma_y = \frac{1}{\max(1/\sigma_i, \|s_i\|_{\text{WRMS}}/|\bar{p}_i|)}$$

or simultaneously:

$$\frac{\partial F}{\partial y}s_i + \frac{\partial F}{\partial y}p\dot{s}_i + \frac{\partial F}{\partial p_i} \approx \frac{F(t, y+\sigma s_i, \dot{y}+\sigma \dot{s}_i, p+\sigma e_i) - F(t, y-\sigma s_i, \dot{y}-\sigma \dot{s}_i, p-\sigma e_i)}{2\,\sigma}\,, \tag{2.16}$$

$$\sigma = \min(\sigma_i, \sigma_y)\,,$$

or by adaptively switching between (2.14) + (2.15) and (2.16), depending on the relative size of the two finite difference increments σ_i and σ_y. In the adaptive scheme, if $\rho = \max(\sigma_i/\sigma_y, \sigma_y/\sigma_i)$, we use separate evaluations if $\rho > \rho_{max}$ (an input value), and simultaneous evaluations otherwise.

These procedures for choosing the perturbations (σ_i, σ_y, σ) and switching between derivative formulas have also been implemented for one-sided difference formulas. Forward finite differences can be applied to $(\partial F/\partial y)s_i + (\partial F/\partial \dot{y})\dot{s}_i$ and $\partial F/\partial p_i$ separately, or the single directional derivative formula

$$\frac{\partial F}{\partial y}s_i + \frac{\partial F}{\partial y}p\dot{s}_i + \frac{\partial F}{\partial p_i} \approx \frac{F(t, y + \sigma s_i, \dot{y} + \sigma \dot{s}_i, p + \sigma e_i) - F(t, y, \dot{y}, p)}{\sigma}$$

can be used. In IDAS, the default value of $\rho_{max} = 0$ indicates the use of the second-order centered directional derivative formula (2.16) exclusively. Otherwise, the magnitude of ρ_{max} and its sign (positive or negative) indicates whether this switching is done with regard to (centered or forward) finite differences, respectively.

2.5.4 Quadratures depending on forward sensitivities

If pure quadrature variables are also included in the problem definition (see §2.4), IDAS does *not* carry their sensitivities automatically. Instead, we provide a more general feature through which integrals depending on both the states y of (2.2) and the state sensitivities s_i of (2.11) can be evaluated. In other words, IDAS provides support for computing integrals of the form:

$$\bar{z}(t) = \int_{t_0}^{t} \bar{q}(\tau, y(\tau), \dot{y}(\tau), s_1(\tau), \ldots, s_{N_p}(\tau), p)\, \mathrm{d}\tau\,.$$

If the sensitivities of the quadrature variables z of (2.10) are desired, these can then be computed by using:

$$\bar{q}_i = q_y s_i + q_{\dot{y}}\dot{s}_i + q_{p_i}\,, \quad i = 1, \ldots, N_p\,,$$

as integrands for $\bar{z}$, where q_y, $q_{\dot{y}}$, and q_p are the partial derivatives of the integrand function q of (2.10).

As with the quadrature variables z, the new variables $\bar{z}$ are also excluded from any nonlinear solver phase and "corrected" values $\bar{z}_n$ are obtained through explicit formulas.

2.6 Adjoint sensitivity analysis

In the *forward sensitivity approach* described in the previous section, obtaining sensitivities with respect to N_s parameters is roughly equivalent to solving an DAE system of size $(1 + N_s)N$. This can become prohibitively expensive, especially for large-scale problems, if sensitivities with respect to many parameters are desired. In this situation, the *adjoint sensitivity method* is a very attractive alternative, provided that we do not need the solution sensitivities s_i, but rather the gradients with respect to model parameters of a relatively few derived functionals of the solution. In other words, if $y(t)$ is the solution of (2.2), we wish to evaluate the gradient $\mathrm{d}G/\mathrm{d}p$ of

$$G(p) = \int_{t_0}^{T} g(t, y, p)\mathrm{d}t\,, \tag{2.17}$$

or, alternatively, the gradient $\mathrm{d}g/\mathrm{d}p$ of the function $g(t, y, p)$ at the final time $t = T$. The function g must be smooth enough that $\partial g/\partial y$ and $\partial g/\partial p$ exist and are bounded.

In what follows, we only sketch the analysis for the sensitivity problem for both G and g. For details on the derivation see [14].

2.6.1 Sensitivity of $G(p)$

We focus first on solving the sensitivity problem for $G(p)$ defined by (2.17). Introducing a Lagrange multiplier λ, we form the augmented objective function

$$I(p) = G(p) - \int_{t_0}^{T} \lambda^* F(t, y, \dot{y}, p) \mathrm{d}t.$$

Since $F(t, y, \dot{y}, p) = 0$, the sensitivity of G with respect to p is

$$\frac{dG}{dp} = \frac{dI}{dp} = \int_{t_0}^{T} (g_p + g_y y_p) \mathrm{d}t - \int_{t_0}^{T} \lambda^* (F_p + F_y y_p + F_{\dot{y}} \dot{y}_p) \mathrm{d}t, \tag{2.18}$$

where subscripts on functions such as F or g are used to denote partial derivatives. By integration by parts, we have

$$\int_{t_0}^{T} \lambda^* F_{\dot{y}} \dot{y}_p \mathrm{d}t = (\lambda^* F_{\dot{y}} y_p)\big|_{t_0}^{T} - \int_{t_0}^{T} (\lambda^* F_{\dot{y}})' y_p \mathrm{d}t,$$

where $(\cdots)'$ denotes the $t-$derivative. Thus equation (2.18) becomes

$$\frac{dG}{dp} = \int_{t_0}^{T} (g_p - \lambda^* F_p)\, \mathrm{d}t - \int_{t_0}^{T} \left[-g_y + \lambda^* F_y - (\lambda^* F_{\dot{y}})'\right] y_p \mathrm{d}t - (\lambda^* F_{\dot{y}} y_p)\big|_{t_0}^{T}.$$

Now by requiring λ to satisfy

$$(\lambda^* F_{\dot{y}})' - \lambda^* F_y = -g_y, \tag{2.19}$$

we obtain

$$\frac{dG}{dp} = \int_{t_0}^{T} (g_p - \lambda^* F_p)\, \mathrm{d}t - (\lambda^* F_{\dot{y}} y_p)\big|_{t_0}^{T}. \tag{2.20}$$

Note that y_p at $t = t_0$ is the sensitivity of the initial conditions with respect to p, which is easily obtained. To find the initial conditions (at $t = T$) for the adjoint system, we must take into consideration the structure of the DAE system.

For index-0 and index-1 DAE systems, we can simply take

$$\lambda^* F_{\dot{y}}\big|_{t=T} = 0, \tag{2.21}$$

yielding the sensitivity equation for dG/dp

$$\frac{dG}{dp} = \int_{t_0}^{T} (g_p - \lambda^* F_p)\, \mathrm{d}t + (\lambda^* F_{\dot{y}} y_p)\big|_{t=t_0}. \tag{2.22}$$

This choice will not suffice for a Hessenberg index-2 DAE system. For a derivation of proper final conditions in such cases, see [14].

The first thing to notice about the adjoint system (2.19) is that there is no explicit specification of the parameters p; this implies that, once the solution λ is found, the formula (2.20) can then be used to find the gradient of G with respect to any of the parameters p. The second important remark is that the adjoint system (2.19) is a terminal value problem which depends on the solution $y(t)$ of the original IVP (2.2). Therefore, a procedure is needed for providing the states y obtained during a forward integration phase of (2.2) to IDAS during the backward integration phase of (2.19). The approach adopted in IDAS, based on *checkpointing*, is described in §2.6.3 below.

2.6.2 Sensitivity of $g(T,p)$

Now let us consider the computation of $\mathrm{d}g/\mathrm{d}p(T)$. From $\mathrm{d}g/\mathrm{d}p(T) = (\mathrm{d}/\mathrm{d}T)(\mathrm{d}G/\mathrm{d}p)$ and equation (2.20), we have

$$\frac{\mathrm{d}g}{\mathrm{d}p} = (g_p - \lambda^* F_p)(T) - \int_{t_0}^{T} \lambda_T^* F_p \mathrm{d}t + (\lambda_T^* F_{\dot{y}} y_p)\Big|_{t=t_0} - \frac{\mathrm{d}(\lambda^* F_{\dot{y}} y_p)}{\mathrm{d}T} \tag{2.23}$$

where λ_T denotes $\partial\lambda/\partial T$. For index-0 and index-1 DAEs, we obtain

$$\frac{\mathrm{d}(\lambda^* F_{\dot{y}} y_p)\big|_{t=T}}{\mathrm{d}T} = 0,$$

while for a Hessenberg index-2 DAE system we have

$$\frac{\mathrm{d}(\lambda^* F_{\dot{y}} y_p)\big|_{t=T}}{\mathrm{d}T} = -\left.\frac{\mathrm{d}(g_{y^a}(CB)^{-1} f_p^2)}{\mathrm{d}t}\right|_{t=T}.$$

The corresponding adjoint equations are

$$(\lambda_T^* F_{\dot{y}})' - \lambda_T^* F_y = 0. \tag{2.24}$$

For index-0 and index-1 DAEs (as shown above, the index-2 case is different), to find the boundary condition for this equation we write λ as $\lambda(t,T)$ because it depends on both t and T. Then

$$\lambda^*(T,T) F_{\dot{y}}\Big|_{t=T} = 0.$$

Taking the total derivative, we obtain

$$(\lambda_t + \lambda_T)^*(T,T) F_{\dot{y}}\Big|_{t=T} + \lambda^*(T,T)\frac{\mathrm{d}F_{\dot{y}}}{\mathrm{d}t}\Big|_{t=T} = 0.$$

Since λ_t is just $\dot{\lambda}$, we have the boundary condition

$$(\lambda_T^* F_{\dot{y}})\big|_{t=T} = -\left[\lambda^*(T,T)\frac{\mathrm{d}F_{\dot{y}}}{\mathrm{d}t} + \dot{\lambda}^* F_{\dot{y}}\right]\Big|_{t=T}.$$

For the index-one DAE case, the above relation and (2.19) yield

$$(\lambda_T^* F_{\dot{y}})\Big|_{t=T} = [g_y - \lambda^* F_y]\Big|_{t=T}.$$

For the regular implicit ODE case, $F_{\dot{y}}$ is invertible; thus we have $\lambda(T,T) = 0$, which leads to $\lambda_T(T) = -\dot{\lambda}(T)$. As with the final conditions for $\lambda(T)$ in (2.19), the above selection for $\lambda_T(T)$ is not sufficient for index-two Hessenberg DAEs (see [14] for details).

2.6.3 Checkpointing scheme

During the backward integration, the evaluation of the right-hand side of the adjoint system requires, at the current time, the states y which were computed during the forward integration phase. Since IDAS implements variable-step integration formulas, it is unlikely that the states will be available at the desired time and so some form of interpolation is needed. The IDAS implementation being also variable-order, it is possible that during the forward integration phase the order may be reduced as low as first order, which means that there may be points in time where only y and $\dot{y}$ are available. These requirements therefore limit the choices for possible interpolation schemes. IDAS implements two interpolation methods: a cubic Hermite interpolation algorithm and a variable-degree polynomial interpolation method which attempts to mimic the BDF interpolant for the forward integration.

However, especially for large-scale problems and long integration intervals, the number and size of the vectors y and $\dot{y}$ that would need to be stored make this approach computationally intractable. Thus, IDAS settles for a compromise between storage space and execution time by implementing a so-called *checkpointing scheme*. At the cost of at most one additional forward integration, this approach offers the best possible estimate of memory requirements for adjoint sensitivity analysis. To begin with, based on the problem size N and the available memory, the user decides on the number N_d of data pairs (y, $\dot{y}$) if cubic Hermite interpolation is selected, or on the number N_d of y vectors in the case of variable-degree polynomial interpolation, that can be kept in memory for the purpose of interpolation. Then, during the first forward integration stage, after every N_d integration steps a checkpoint is formed by saving enough information (either in memory or on disk) to allow for a hot restart, that is a restart which will exactly reproduce the forward integration. In order to avoid storing Jacobian-related data at each checkpoint, a reevaluation of the iteration matrix is forced before each checkpoint. At the end of this stage, we are left with N_c checkpoints, including one at t_0. During the backward integration stage, the adjoint variables are integrated backwards from T to t_0, going from one checkpoint to the previous one. The backward integration from checkpoint $i+1$ to checkpoint i is preceded by a forward integration from i to $i+1$ during which the N_d vectors y (and, if necessary $\dot{y}$) are generated and stored in memory for interpolation.

Note: The degree of the interpolation polynomial is always that of the current BDF order for the forward interpolation at the first point to the right of the time at which the interpolated value is sought (unless too close to the i-th checkpoint, in which case it uses the BDF order at the right-most relevant point). However, because of the FLC BDF implementation (see §2.1), the resulting interpolation polynomial is only an approximation to the underlying BDF interpolant.

The Hermite cubic interpolation option is present because it was implemented chronologically first and it is also used by other adjoint solvers (e.g. `DASPKADJOINT`). The variable-degree polynomial is more memory-efficient (it requires only half of the memory storage of the cubic Hermite interpolation) and is more accurate.

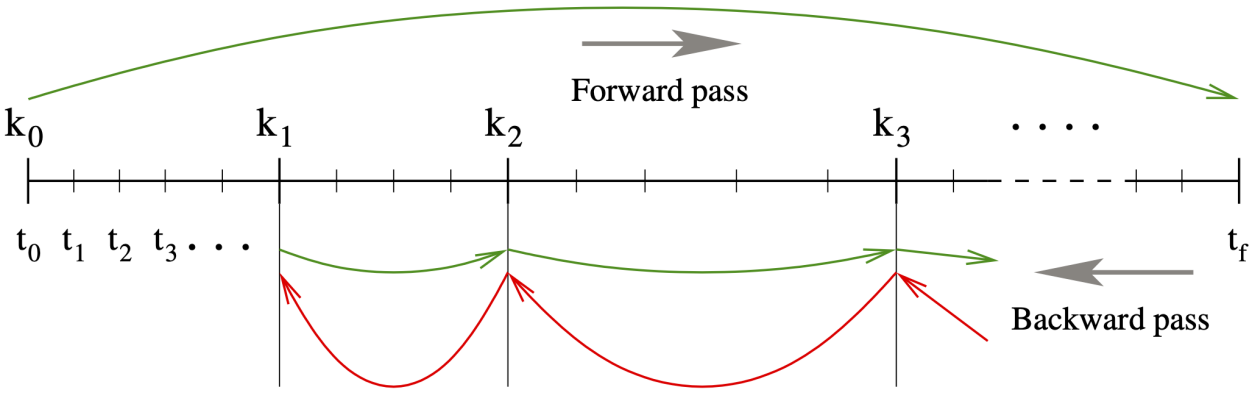

Fig. 2.1: Illustration of the checkpointing algorithm for generation of the forward solution during the integration of the adjoint system.

This approach transfers the uncertainty in the number of integration steps in the forward integration phase to uncertainty in the final number of checkpoints. However, N_c is much smaller than the number of steps taken during the forward integration, and there is no major penalty for writing/reading the checkpoint data to/from a temporary file. Note that, at the end of the first forward integration stage, interpolation data are available from the last checkpoint to the end of the interval of integration. If no checkpoints are necessary (N_d is larger than the number of integration steps taken in the solution of (2.2)), the total cost of an adjoint sensitivity computation can be as low as one forward plus one backward integration. In addition, IDAS provides the capability of reusing a set of checkpoints for multiple backward integrations, thus allowing for efficient computation of gradients of several functionals (2.17).

Finally, we note that the adjoint sensitivity module in IDAS provides the necessary infrastructure to integrate backwards in time any DAE terminal value problem dependent on the solution of the IVP (2.2), including adjoint systems (2.19) or (2.24), as well as any other quadrature ODEs that may be needed in evaluating the integrals in (2.20). In particular,

for DAE systems arising from semi-discretization of time-dependent PDEs, this feature allows for integration of either the discretized adjoint PDE system or the adjoint of the discretized PDE.

2.7 Second-order sensitivity analysis

In some applications (e.g., dynamically-constrained optimization) it may be desirable to compute second-order derivative information. Considering the DAE problem (2.2) and some model output functional $g(y)$, the Hessian $\mathrm{d}^2 g/\mathrm{d}p^2$ can be obtained in a forward sensitivity analysis setting as

$$\frac{\mathrm{d}^2 g}{\mathrm{d}p^2} = \left(g_y \otimes I_{N_p}\right) y_{pp} + y_p^T g_{yy} y_p \,,$$

where $\otimes$ is the Kronecker product. The second-order sensitivities are solution of the matrix DAE system:

$$\begin{aligned}
&\left(F_{\dot{y}} \otimes I_{N_p}\right) \cdot \dot{y}_{pp} + \left(F_y \otimes I_{N_p}\right) \cdot y_{pp} + \left(I_N \otimes \dot{y}_p^T\right) \cdot \left(F_{\dot{y}\dot{y}} \dot{y}_p + F_{y\dot{y}} y_p\right) + \left(I_N \otimes y_p^T\right) \cdot \left(F_{y\dot{y}} \dot{y}_p + F_{yy} y_p\right) = 0 \\
&y_{pp}(t_0) = \frac{\partial^2 y_0}{\partial p^2} \,, \quad \dot{y}_{pp}(t_0) = \frac{\partial^2 \dot{y}_0}{\partial p^2} \,,
\end{aligned}$$

where y_p denotes the first-order sensitivity matrix, the solution of N_p systems (2.11), and y_{pp} is a third-order tensor. It is easy to see that, except for situations in which the number of parameters N_p is very small, the computational cost of this so-called *forward-over-forward* approach is exorbitant as it requires the solution of $N_p + N_p^2$ additional DAE systems of the same dimension as (2.2).

Note: For the sake of simplifity in presentation, we do not include explicit dependencies of g on time t or parameters p. Moreover, we only consider the case in which the dependency of the original DAE (2.2) on the parameters p is through its initial conditions only. For details on the derivation in the general case, see [43].

A much more efficient alternative is to compute Hessian-vector products using a so-called *forward-over-adjoint* approach. This method is based on using the same "trick" as the one used in computing gradients of pointwise functionals with the adjoint method, namely applying a formal directional forward derivation to the gradient of (2.20) (or the equivalent one for a pointwise functional $g(T, y(T))$). With that, the cost of computing a full Hessian is roughly equivalent to the cost of computing the gradient with forward sensitivity analysis. However, Hessian-vector products can be cheaply computed with one additional adjoint solve.

As an illustration, consider the ODE problem (the derivation for the general DAE case is too involved for the purposes of this discussion)

$$\dot{y} = f(t, y) \,, \quad y(t_0) = y_0(p) \,,$$

depending on some parameters p through the initial conditions only and consider the model functional output $G(p) = \int_{t_0}^{t_f} g(t, y)\, \mathrm{d}t$. It can be shown that the product between the Hessian of G (with respect to the parameters p) and some vector u can be computed as

$$\frac{\partial^2 G}{\partial p^2} u = \left[\left(\lambda^T \otimes I_{N_p}\right) y_{pp} u + y_p^T \mu\right]_{t=t_0} \,,$$

where λ and μ are solutions of

$$\begin{aligned}
-\dot{\mu} &= f_y^T \mu + \left(\lambda^T \otimes I_n\right) f_{yy} s \,; \quad \mu(t_f) = 0 \\
-\dot{\lambda} &= f_y^T \lambda + g_y^T \,; \quad \lambda(t_f) = 0 \\
\dot{s} &= f_y s \,; \quad s(t_0) = y_{0p} u.
\end{aligned}$$

In the above equation, $s = y_p u$ is a linear combination of the columns of the sensitivity matrix y_p. The *forward-over-adjoint* approach hinges crucially on the fact that s can be computed at the cost of a forward sensitivity analysis

with respect to a single parameter (the last ODE problem above) which is possible due to the linearity of the forward sensitivity equations (2.11).

Therefore (and this is also valid for the DAE case), the cost of computing the Hessian-vector product is roughly that of two forward and two backward integrations of a system of DAEs of size N. For more details, including the corresponding formulas for a pointwise model functional output, see the work by Ozyurt and Barton [43] who discuss this problem for ODE initial value problems. As far as we know, there is no published equivalent work on DAE problems. However, the derivations given in [43] for ODE problems can be extended to DAEs with some careful consideration given to the derivation of proper final conditions on the adjoint systems, following the ideas presented in [14].

To allow the *foward-over-adjoint* approach described above, IDAS provides support for:

- the integration of multiple backward problems depending on the same underlying forward problem (2.2), and
- the integration of backward problems and computation of backward quadratures depending on both the states y and forward sensitivities (for this particular application, s) of the original problem (2.2).

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 IDAS organization

The IDAS 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 IDAS package is shown in Fig. 3.3. IDAS utilizes generic linear and nonlinear solvers defined by the `SUNLinearSolver` (see §8) and `SUNNonlinearSolver` interfaces (see §9) respectively. As such, IDAS 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.

IDAS has a single unified linear solver interface, IDALS, 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

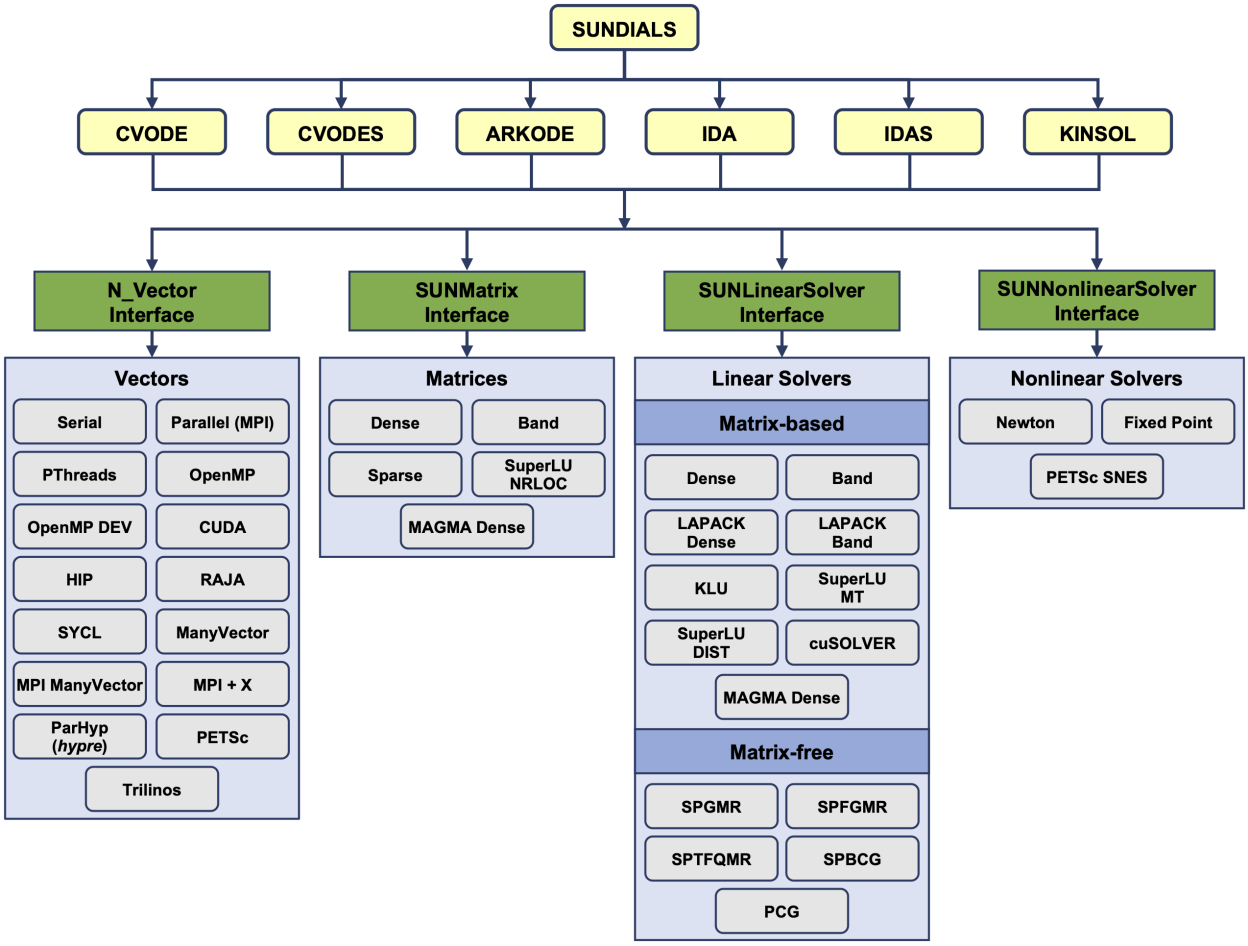

Fig. 3.1: High-level diagram of the SUNDIALS suite.

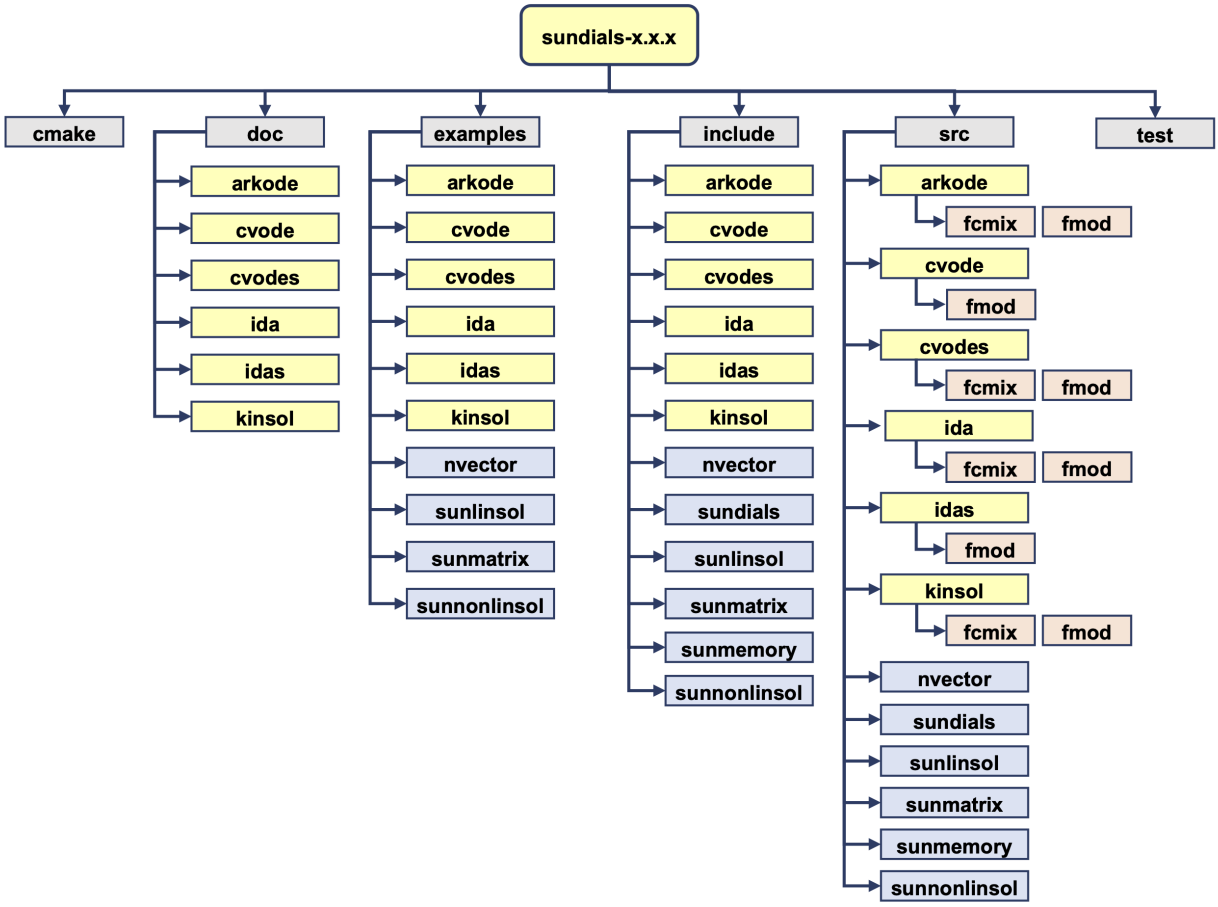

Fig. 3.2: Directory structure of the SUNDIALS source tree.

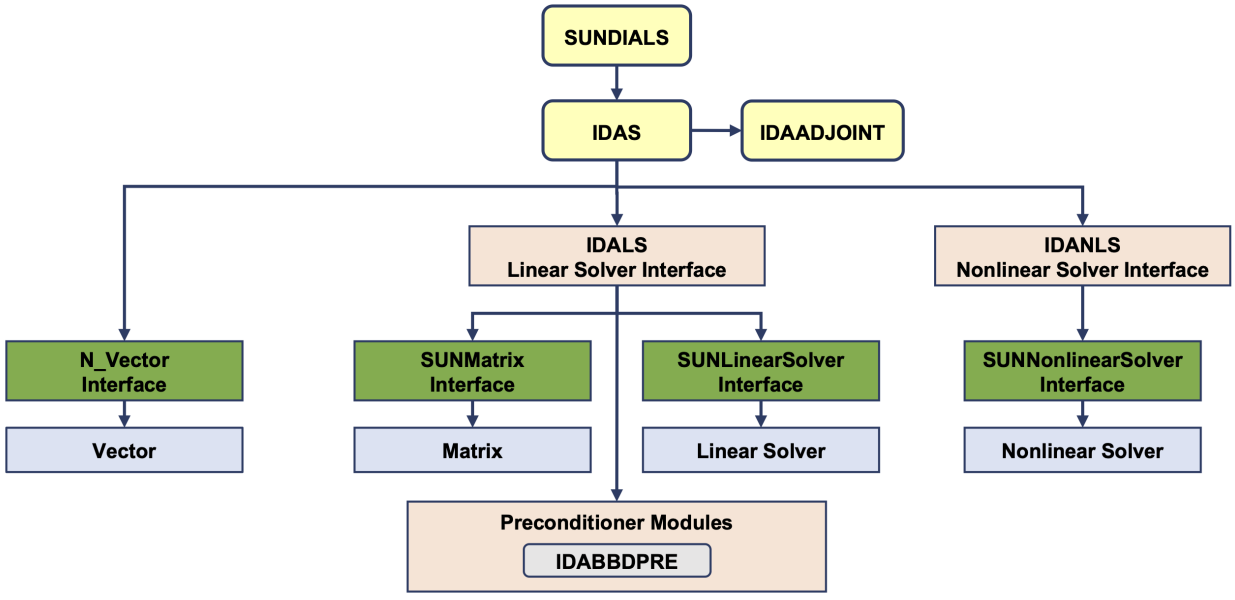

Fig. 3.3: Overall structure diagram of the IDAS package. Components specific to IDAS begin with "IDA" (IDALS, IDANLS, and IDABBDPRE), all other items correspond to generic SUNDIALS vector, matrix, and solver interfaces.

Jacobian information, or they may be matrix-free. Since IDAS can operate on any valid `SUNLinearSolver`, the set of linear solver modules available to IDAS will expand as new `SUNLinearSolver` implementations are developed.

For users employing *SUNMATRIX_DENSE* or *SUNMATRIX_BAND* Jacobian matrices, IDAS 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, IDAS 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 [7, 11], together with the example and demonstration programs included with IDAS, offer considerable assistance in building preconditioners.

IDA'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 IDAS module to each of the four associated functions is fixed, thus allowing the central module to be completely independent of the linear system method.

IDAS 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 IDAS to solve a given problem is stored in `N_Vector` instances. There is no global data in the IDAS 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 IDAS 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 all SUNDIALS packages are built upon a common set of interfaces for vectors, matrices, and algebraic solvers. In addition, the packages all leverage some other common infrastructure discussed in this section.

4.1 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

4.1.1 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 §11.1.2).

4.1.2 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 §11.1.2).

4.1.3 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`).

4.2 The SUNContext Type

In SUNDIALS v6.0.0, the concept of a SUNDIALS context object was introduced. Every SUNDIALS object holds a reference to a common `SUNContext` object, which is in turn unique to a thread of execution.

The `SUNContext` 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);
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.

4.2.1 Implications for Multi-Threading

In multi-threading applications where multiple SUNDIALS simulations are conducted concurrently, e.g. by having one instance of an integrator per thread, a `SUNContext` object needs to be created for each thread.

4.2.2 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_;

};

} // namespace sundials
```

4.3 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 §4.3.2).

4.3.1 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 §11.

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.

4.3.2 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)
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

Returns:

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

4.3.3 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   */
```

4.3.4 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`.

4.4 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*.

4.5 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*, *SUNLinearSolver*, and *SUNNonlinearSolver*) 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 §4.5.2. A discussion on the topic of equivalent data types in C and Fortran 2003 is presented in §4.5.1.

Further information on the Fortran 2003 interfaces specific to the *N_Vector*, *SUNMatrix*, *SUNLinearSolver*, and *SUNNonlinearSolver* classes is given alongside the C documentation (§6, §7, §8, and §9 respectively). For details on where the Fortran 2003 module (`.mod`) files and libraries are installed see §11.

The Fortran 2003 interface modules were generated with SWIG Fortran [35], 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
SUNLINSOL_KLU	`fsunlinsol_klu_mod`
SUNLINSOL_SLUMT	Not interfaced
SUNLINSOL_SLUDIST	Not interfaced
SUNLINSOL_SPGMR	`fsunlinsol_spgmr_mod`

continues on next page

Table 4.1 – continued from previous page

Class/Module	Fortran 2003 Module Name
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

4.5.1 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)
SUNMatrix	return	type(SUNMatrix), pointer
SUNLinearSolver	in, inout, out	type(SUNLinearSolver)
SUNLinearSolver	return	type(SUNLinearSolver), pointer
SUNNonlinearSolver	in, inout, out	type(SUNNonlinearSolver)

continues on next page

Table 4.2 – continued from previous page

C Type	Parameter Direction	Fortran 2003 type
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)

4.5.2 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, §4.5.1 discusses equivalencies of data types in the two languages.

4.5.2.1 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.

4.5.2.2 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;

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

(continues on next page)

(continued from previous page)

```
/* 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)
```

4.5.2.3 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

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.

4.5.2.4 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 §4.5.1), 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)
```

4.5.2.5 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 */
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.

4.5.2.6 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)`.

4.5.3 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.

4.6 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, §8, and §9) 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.

4.6.1 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 [52], AMD ROCm/HIP [49], and Intel oneAPI [50]. Table 4.3–Table 4.6 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*, *SUNLinearSolver*, or *SUNNonlinearSolver* 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 §10) 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]

Table 4.6: List of SUNDIALS GPU-enabled `SUNNonlinearSolver` Modules

Module	CUDA	ROCm/HIP	oneAPI	Unmanaged Memory	UVM
SUNNONLINSOL_NEWTON	X[1]	X[1]	X[1]	X[1]	X[1]
SUNNONLINSOL_FIXEDPOINT	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.

4.6.2 Steps for Using GPU Accelerated SUNDIALS

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

1. Utilize a GPU-enabled `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 IDAS

5.1 Using IDAS for IVP Solution

This chapter is concerned with the use of IDAS for the integration of DAEs.

The following sections treat the header files and the layout of the user's main program, and provide descriptions of the IDAS user-callable functions and user-supplied functions. The sample programs described in the companion document [32] 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 IDAS package.

IDAS uses various constants for both input and output. These are defined as needed in this chapter, but for convenience are also listed separately in §12.

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 IDABBDPRE 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.

5.1.1 Access to library and header files

At this point, it is assumed that the installation of IDAS, following the procedure described in §11, 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 IDAS. The relevant library files are

```
<libdir>/libsundials_ida.<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>/idas
<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 §11).

Note that an application cannot link to both the IDAS and IDA libraries because both contain user-callable functions with the same names (to ensure that IDAS is backward compatible with IDA). Therefore, applications that contain both DAE problems and DAEs with sensitivity analysis, should use IDAS.

5.1.2 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:

- `idas/idas.h` the main header file for IDAS, which defines the types and various constants, and includes function prototypes. This includes the header file for IDALS, `idas/idas_ls.h`.

Note that `idas.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 Chapter §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 non-default nonlinear solver object, or when interacting with a `SUNNonlinearSolver` object directly, the calling program must also include a `SUNNonlinearSolver` implementation header file, of the form `sunnonlinsol/sunnonlinsol_*.h` where * is the name of the nonlinear solver (see Chapter §9 for more information). This file in turn includes the header file `sundials_nonlinearsolver.h` which defines the abstract nonlinear linear solver data type.

If using a nonlinear solver that requires the solution of a linear system of the form (2.4) (e.g., the default Newton iteration), the calling program must also include a `SUNLinearSolver` implementation header file, of the from `sunlinsol/sunlinsol_*.h` where * is the name of the linear solver (see Chapter §8 for more information). This file in turn includes the header file `sundials_linearsolver.h` which defines the abstract linear solver data type.

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 `idasFoodWeb_kry_p` (see [32]), 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.

5.1.3 A skeleton of the user's main program

The following is a skeleton of the user's main program (or calling program) for the integration of a DAE IVP. Most of the steps are independent of the `N_Vector`, `SUNMatrix`, `SUNLinearSolver`, and `SUNNonlinearSolver` implementations used. For the steps that are not, refer to Chapters §6, §7, §8, and §9 for the specific name of the function to be called or macro to be referenced.

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

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

2. **Create the SUNDIALS context object**

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

3. **Create the vector of initial values**

   Construct an `N_Vector` of initial 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.

   Set the vector `yp0` of initial conditions for $\dot{y}$ similarly.

4. **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).

5. **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).

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

   If using a non-default nonlinear solver, then the desired nonlinear solver object must be created by calling the appropriate constructor defined by the particular `SUNNonlinearSolver` implementation.

   For any of the native SUNDIALS `SUNNonLinearSolver` implementations, the nonlinear solver object may be created using a call of the form `SUNNonlinearSolver NLS = SUNNonlinSol_***(...);` where `***` is the name of the nonlinear solver (see §9 for details).

7. **Create IDAS object**

   Call *IDACreate()* to create the IDAS solver object.

8. **Initialize IDAS solver**

   Call *IDAInit()* to provide the initial condition vectors created above, set the DAE residual function, and initialize IDAS.

9. **Specify integration tolerances**

   Call one of the following functions to set the integration tolerances:

   - *IDASStolerances()* to specify scalar relative and absolute tolerances.
   - *IDASVtolerances()* to specify a scalar relative tolerance and a vector of absolute tolerances.
   - *IDAWFtolerances()* to specify a function which sets directly the weights used in evaluating WRMS vector norms.

   See §5.1.4.3 for general advice on selecting tolerances and §5.1.4.4 for advice on controlling unphysical values.

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

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

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

    See Table 5.2 for IDALS optional inputs and Chapter §8 for linear solver specific optional inputs.

12. **Attach nonlinear solver module** (*if appropriate*)

    If a nonlinear solver was created above, initialize the IDANLS nonlinear solver interface by attaching the nonlinear solver object with *IDASetNonlinearSolver()*.

13. **Set nonlinear solver optional inputs** (*if appropriate*)

    See Table 5.3 for IDANLS optional inputs and Chapter §9 for nonlinear solver specific optional inputs. Note, solver specific optional inputs *must* be called after *IDASetNonlinearSolver()*, otherwise the optional inputs will be overridden by IDAS defaults.

14. **Specify rootfinding problem** (*optional*)

    Call *IDARootInit()* to initialize a rootfinding problem to be solved during the integration of the ODE system. See Table 5.5 for relevant optional input calls.

15. **Set optional inputs**

    Call `IDASet***` functions to change any optional inputs that control the behavior of IDAS from their default values. See §5.1.4.10 for details.

16. **Correct initial values** (*optional*)

    Call *IDACalcIC()* to correct the initial values `y0` and `yp0` passed to *IDAInit()*. See Table 5.4 for relevant optional input calls.

17. **Advance solution in time**

    For each point at which output is desired, call `ier = IDASolve(ida_mem, tout, &tret, yret, ypret, itask)`. Here `itask` specifies the return mode. The vector `yret` (which can be the same as the vector `y0` above) will contain $y(t)$, while the vector `ypret` (which can be the same as the vector `yp0` above) will contain $\dot{y}(t)$.

    See *IDASolve()* for details.

18. **Get optional outputs**

    Call `IDAGet***` functions to obtain optional output. See §5.1.4.12 for details.

19. **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 *IDAFree()* to free the memory allocated by IDAS.
    - Call *SUNContext_Free()* to free the SUNDIALS context.

20. **Finalize MPI, if used**

    Call `MPI_Finalize` to terminate MPI.

5.1.4 User-callable functions

This section describes the IDAS functions that are called by the user to setup and then solve an IVP. Some of these are required. However, starting with §5.1.4.10, the functions listed involve optional inputs/outputs or restarting, and those paragraphs may be skipped for a casual use of IDAS. In any case, refer to §5.1.3 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 §5.1.4.10).

5.1.4.1 IDAS initialization and deallocation functions

void ***IDACreate**(*SUNContext* sunctx)
The function *IDACreate()* instantiates an IDAS solver object.

Arguments:

- `sunctx` – the *SUNContext* object (see §4.2)

Return value:

- `void*` pointer the IDAS solver object.

int **IDAInit**(void *ida_mem, *IDAResFn* res, *realtype* t0, *N_Vector* y0, *N_Vector* yp0)
The function *IDAInit()* provides required problem and solution specifications, allocates internal memory, and initializes IDAS.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `res` – is the function which computes the residual function $F(t, y, \dot{y})$ for the DAE. For full details see *IDAResFn*.
- `t0` – is the initial value of t.
- `y0` – is the initial value of y.
- `yp0` – is the initial value of $\dot{y}$.

Return value:

- `IDA_SUCCESS` – The call was successful.

- IDA_MEM_NULL – The ida_mem argument was NULL.
- IDA_MEM_FAIL – A memory allocation request has failed.
- IDA_ILL_INPUT – An input argument to *IDAInit()* has an illegal value.

Notes: If an error occurred, *IDAInit()* also sends an error message to the error handler function.

void **IDAFree**(void **ida_mem)
The function *IDAFree()* frees the pointer allocated by a previous call to *IDACreate()*.

Arguments:

- ida_mem – pointer to the IDAS solver object.

Return value:

- void

5.1.4.2 IDAS tolerance specification functions

One of the following three functions must be called to specify the integration tolerances (or directly specify the weights used in evaluating WRMS vector norms). Note that this call must be made after the call to *IDAInit()*.

int **IDASStolerances**(void *ida_mem, *realtype* reltol, *realtype* abstol)
The function *IDASStolerances()* specifies scalar relative and absolute tolerances.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- reltol – is the scalar relative error tolerance.
- abstol – is the scalar absolute error tolerance.

Return value:

- IDA_SUCCESS – The call was successful.
- IDA_MEM_NULL – The ida_mem argument was NULL.
- IDA_NO_MALLOC – The allocation function *IDAInit()* has not been called.
- IDA_ILL_INPUT – One of the input tolerances was negative.

int **IDASVtolerances**(void *ida_mem, *realtype* reltol, *N_Vector* abstol)
The function *IDASVtolerances()* specifies scalar relative tolerance and vector absolute tolerances.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- reltol – is the scalar relative error tolerance.
- abstol – is the vector of absolute error tolerances.

Return value:

- IDA_SUCCESS – The call was successful.
- IDA_MEM_NULL – The ida_mem argument was NULL.
- IDA_NO_MALLOC – The allocation function *IDAInit()* has not been called.
- IDA_ILL_INPUT – The relative error tolerance was negative or the absolute tolerance vector had a negative component.

Notes: This choice of tolerances is important when the absolute error tolerance needs to be different for each component of the state vector y.

int **IDAWFtolerances**(void *ida_mem, *IDAEwtFn* efun)

The function *IDAWFtolerances()* specifies a user-supplied function `efun` that sets the multiplicative error weights W_i for use in the weighted RMS norm, which are normally defined by (2.7).

Arguments:

- `ida_mem` – pointer to the IDAS solver object. *IDACreate()*
- `efun` – is the function which defines the `ewt` vector. For full details see *IDAEwtFn*.

Return value:

- `IDA_SUCCESS` – The call was successful.
- `IDA_MEM_NULL` – The `ida_mem` argument was `NULL`.
- `IDA_NO_MALLOC` – The allocation function *IDAInit()* has not been called.

5.1.4.3 General advice on choice of tolerances

For many users, the appropriate choices for tolerance values in `reltol` and `abstol` are a concern. The following pieces of advice are relevant.

1. The scalar relative tolerance `reltol` is to be set to control relative errors. So `reltol` of 10^{-4} means that errors are controlled to .01%. We do not recommend using `reltol` larger than 10^{-3}. On the other hand, `reltol` should not be so small that it is comparable to the unit roundoff of the machine arithmetic (generally around 10^{-15}).
2. The absolute tolerances `abstol` (whether scalar or vector) need to be set to control absolute errors when any components of the solution vector `y` may be so small that pure relative error control is meaningless. For example, if `y[i]` starts at some nonzero value, but in time decays to zero, then pure relative error control on `y[i]` makes no sense (and is overly costly) after `y[i]` is below some noise level. Then `abstol` (if a scalar) or `abstol[i]` (if a vector) needs to be set to that noise level. If the different components have different noise levels, then `abstol` should be a vector. See the example `idaRoberts_dns` in the IDAS package, and the discussion of it in the IDAS Examples document [32]. In that problem, the three components vary betwen 0 and 1, and have different noise levels; hence the `abstol` vector. It is impossible to give any general advice on `abstol` values, because the appropriate noise levels are completely problem-dependent. The user or modeler hopefully has some idea as to what those noise levels are.
3. Finally, it is important to pick all the tolerance values conservatively, because they control the error committed on each individual time step. The final (global) errors are some sort of accumulation of those per-step errors. A good rule of thumb is to reduce the tolerances by a factor of .01 from the actual desired limits on errors. So if you want .01% accuracy (globally), a good choice is to is a `reltol` of 10^{-6}. But in any case, it is a good idea to do a few experiments with the tolerances to see how the computed solution values vary as tolerances are reduced.

5.1.4.4 Advice on controlling unphysical negative values

In many applications, some components in the true solution are always positive or non-negative, though at times very small. In the numerical solution, however, small negative (hence unphysical) values can then occur. In most cases, these values are harmless, and simply need to be controlled, not eliminated. The following pieces of advice are relevant.

1. The way to control the size of unwanted negative computed values is with tighter absolute tolerances. Again this requires some knowledge of the noise level of these components, which may or may not be different for different components. Some experimentation may be needed.

2. If output plots or tables are being generated, and it is important to avoid having negative numbers appear there (for the sake of avoiding a long explanation of them, if nothing else), then eliminate them, but only in the context of the output medium. Then the internal values carried by the solver are unaffected. Remember that a small negative value in `yret` returned by IDAS, with magnitude comparable to `abstol` or less, is equivalent to zero as far as the computation is concerned.

3. The user's residual function `res` should never change a negative value in the solution vector `yy` to a non-negative value, as a "solution" to this problem. This can cause instability. If the `res` routine cannot tolerate a zero or negative value (e.g., because there is a square root or log of it), then the offending value should be changed to zero or a tiny positive number in a temporary variable (not in the input `yy` vector) for the purposes of computing $F(t, y, \dot{y})$.

4. IDAS provides the option of enforcing positivity or non-negativity on components. Also, such constraints can be enforced by use of the recoverable error return feature in the user-supplied residual function. However, because these options involve some extra overhead cost, they should only be exercised if the use of absolute tolerances to control the computed values is unsuccessful.

5.1.4.5 Linear solver interface functions

As previously explained, if the nonlinear solver requires the solution of linear systems of the form (2.5), e.g., the default Newton solver, then the solution of these linear systems is handled with the IDALS linear solver interface. This interface supports all valid `SUNLinearSolver` objects. Here, a matrix-based `SUNLinearSolver` utilizes `SUNMatrix` objects to store the Jacobian matrix $J = \dfrac{\partial F}{\partial y} + \alpha \dfrac{\partial F}{\partial \dot{y}}$ and factorizations used throughout the solution process. Conversely, matrix-free `SUNLinearSolver` object 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. The exceptions to this rule are SPFGMR that supports right preconditioning only and PCG that performs symmetric preconditioning. However, in IDAS only left preconditioning is supported. For the specification of a preconditioner, see the iterative linear solver sections in §5.1.4.10 and §5.1.5. A preconditioner matrix P must approximate the Jacobian J, at least crudely.

To attach a generic linear solver to IDAS, after the call to *IDACreate()* but before any calls to *IDASolve()*, the user's program must create the appropriate `SUNLinearSolver` object and call the function *IDASetLinearSolver()*. To create the `SUNLinearSolver` object, the user may call one of the SUNDIALS-packaged `SUNLinearSolver` constructors via a call of the form

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

Alternately, a user-supplied `SUNLinearSolver` object 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` object in question, as described in Chapters §7 and §8.

Once this solver object has been constructed, the user should attach it to IDAS via a call to *IDASetLinearSolver()*. The first argument passed to this function is the IDAS memory pointer returned by *IDACreate()*; the second argument is the desired `SUNLinearSolver` object to use for solving 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 IDALS linear solver interface, linking it to the main IDAS integrator, and allows the user to specify additional parameters and routines pertinent to their choice of linear solver.

int **IDASetLinearSolver**(void *ida_mem, *SUNLinearSolver* LS, *SUNMatrix* J)

The function *IDASetLinearSolver()* attaches a `SUNLinearSolver` object `LS` and corresponding template Jacobian `SUNMatrix` object `J` (if applicable) to IDAS, initializing the IDALS linear solver interface.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- LS – SUNLinearSolver object to use for solving linear systems of the form (2.5).
- J – SUNMatrix object for used as a template for the Jacobian or NULL if not applicable.

Return value:

- IDALS_SUCCESS – The IDALS initialization was successful.
- IDALS_MEM_NULL – The ida_mem pointer is NULL.
- IDALS_ILL_INPUT – The IDALS interface is not compatible with the LS or J input objects or is incompatible with the N_Vector object passed to *IDAInit()*.
- IDALS_SUNLS_FAIL – A call to the LS object failed.
- IDALS_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 in Chapter §7 for further information).

Warning: The previous routines IDADlsSetLinearSolver() and IDASpilsSetLinearSolver() 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.

5.1.4.6 Nonlinear solver interface function

By default IDAS uses the SUNNonlinearSolver implementation of Newton's method (see §9.3). To attach a different nonlinear solver in IDAS, the user's program must create a SUNNonlinearSolver object by calling the appropriate constructor routine. The user must then attach the SUNNonlinearSolver object to IDAS by calling *IDASetNonlinearSolver()*.

When changing the nonlinear solver in IDAS, *IDASetNonlinearSolver()* must be called after *IDAInit()*. If any calls to *IDASolve()* have been made, then IDAS will need to be reinitialized by calling *IDAReInit()* to ensure that the nonlinear solver is initialized correctly before any subsequent calls to *IDASolve()*.

The first argument passed to *IDASetNonlinearSolver()* is the IDAS memory pointer returned by *IDACreate()* and the second argument is the SUNNonlinearSolver object to use for solving the nonlinear system (2.4). A call to this function attaches the nonlinear solver to the main IDAS integrator. We note that at present, the SUNNonlinearSolver object *must be of type* SUNNONLINEARSOLVER_ROOTFIND.

int **IDASetNonlinearSolver**(void *ida_mem, *SUNNonlinearSolver* NLS)

The function IDASetNonLinearSolver() attaches a SUNNonlinearSolver object (NLS) to IDAS.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- NLS – SUNNonlinearSolver object to use for solving nonlinear systems.

Return value:

- IDA_SUCCESS – The nonlinear solver was successfully attached.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

- IDA_ILL_INPUT – The SUNNonlinearSolver object is NULL , does not implement the required nonlinear solver operations, is not of the correct type, or the residual function, convergence test function, or maximum number of nonlinear iterations could not be set.

Notes: When forward sensitivity analysis capabilities are enabled and the IDA_STAGGERED corrector method is used this function sets the nonlinear solver method for correcting state variables (see §5.4.2.3 for more details).

5.1.4.7 Initial condition calculation function

IDACalcIC() calculates corrected initial conditions for the DAE system for certain index-one problems including a class of systems of semi-implicit form (see §2.1 and [9]). It uses a Newton iteration combined with a linesearch algorithm. Calling *IDACalcIC()* is optional. It is only necessary when the initial conditions do not satisfy the given system. Thus if y0 and yp0 are known to satisfy $F(t_0, y_0, \dot{y}_0) = 0$, then a call to *IDACalcIC()* is generally *not* necessary.

A call to the function *IDACalcIC()* must be preceded by successful calls to *IDACreate()* and *IDAInit()* (or *IDAReInit()*), and by a successful call to the linear system solver specification function. The call to *IDACalcIC()* should precede the call(s) to *IDASolve()* for the given problem.

int **IDACalcIC**(void *ida_mem, int icopt, *realtype* tout1)

The function *IDACalcIC()* corrects the initial values y0 and yp0 at time t0.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- icopt – is one of the following two options for the initial condition calculation.
  - IDA_YA_YDP_INIT directs *IDACalcIC()* to compute the algebraic components of y and differential components of $\dot{y}$, given the differential components of y. This option requires that the N_Vector id was set through *IDASetId()*, specifying the differential and algebraic components.
  - IDA_Y_INIT directs *IDACalcIC()* to compute all components of y, given $\dot{y}$. In this case, id is not required.
- tout1 – is the first value of t at which a solution will be requested (from *IDASolve()*). This value is needed here only to determine the direction of integration and rough scale in the independent variable t.

Return value:

- IDA_SUCCESS – *IDACalcIC()* succeeded.
- IDA_MEM_NULL – The argument ida_mem was NULL.
- IDA_NO_MALLOC – The allocation function *IDAInit()* has not been called.
- IDA_ILL_INPUT – One of the input arguments was illegal.
- IDA_LSETUP_FAIL – The linear solver's setup function failed in an unrecoverable manner.
- IDA_LINIT_FAIL – The linear solver's initialization function failed.
- IDA_LSOLVE_FAIL – The linear solver's solve function failed in an unrecoverable manner.
- IDA_BAD_EWT – Some component of the error weight vector is zero (illegal), either for the input value of y0 or a corrected value.
- IDA_FIRST_RES_FAIL – The user's residual function returned a recoverable error flag on the first call, but *IDACalcIC()* was unable to recover.
- IDA_RES_FAIL – The user's residual function returned a nonrecoverable error flag.

- IDA_NO_RECOVERY – The user's residual function, or the linear solver's setup or solve function had a recoverable error, but *IDACalcIC()* was unable to recover.
- IDA_CONSTR_FAIL – *IDACalcIC()* was unable to find a solution satisfying the inequality constraints.
- IDA_LINESEARCH_FAIL – The linesearch algorithm failed to find a solution with a step larger than steptol in weighted RMS norm, and within the allowed number of backtracks.
- IDA_CONV_FAIL – *IDACalcIC()* failed to get convergence of the Newton iterations.

Notes: *IDACalcIC()* will correct the values of $y(t_0)$ and $\dot{y}(t_0)$ which were specified in the previous call to *IDAInit()* or *IDAReInit()*. To obtain the corrected values, call *IDAGetConsistentIC()*.

5.1.4.8 Rootfinding initialization function

While solving the IVP, IDAS has the capability to find the roots of a set of user-defined functions. To activate the root finding algorithm, call the following function. This is normally called only once, prior to the first call to *IDASolve()*, but if the rootfinding problem is to be changed during the solution, *IDARootInit()* can also be called prior to a continuation call to *IDASolve()*.

int **IDARootInit**(void *ida_mem, int nrtfn, *IDARootFn* g)

The function *IDARootInit()* specifies that the roots of a set of functions $g_i(t, y)$ are to be found while the IVP is being solved.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- nrtfn – is the number of root functions.
- g – is the function which defines the nrtfn functions $g_i(t, y, \dot{y})$ whose roots are sought. See *IDARootFn* for more details.

Return value:

- IDA_SUCCESS – The call was successful.
- IDA_MEM_NULL – The ida_mem argument was NULL.
- IDA_MEM_FAIL – A memory allocation failed.
- IDA_ILL_INPUT – The function g is NULL, but nrtfn > 0.

Notes: If a new IVP is to be solved with a call to *IDAReInit()*, where the new IVP has no rootfinding problem but the prior one did, then call *IDARootInit()* with nrtfn = 0.

5.1.4.9 IDAS solver function

This is the central step in the solution process, the call to perform the integration of the DAE. The input arguments (itask) specifies one of two modes as to where IDAS is to return a solution. These modes are modified if the user has set a stop time (with *IDASetStopTime()*) or requested rootfinding (with *IDARootInit()*).

int **IDASolve**(void *ida_mem, *realtype* tout, *realtype* *tret, *N_Vector* yret, *N_Vector* ypret, int itask)

The function *IDASolve()* integrates the DAE over an interval in t.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- tout – the next time at which a computed solution is desired.
- tret – the time reached by the solver output.

- `yret` – the computed solution vector y.
- `ypret` – the computed solution vector $\dot{y}$.
- `itask` – a flag indicating the job of the solver for the next user step
  - `IDA_NORMAL` – the solver will take internal steps until it has reached or just passed the user specified `tout` parameter. The solver then interpolates in order to return approximate values of $y(t_{out})$ and $\dot{y}(t_{out})$.
  - `IDA_ONE_STEP` – the solver will just take one internal step and return the solution at the point reached by that step.

Return value:

- `IDA_SUCCESS` – The call was successful.
- `IDA_TSTOP_RETURN` – *IDASolve()* succeeded by reaching the stop point specified through the optional input function *IDASetStopTime()*.
- `IDA_ROOT_RETURN` – *IDASolve()* succeeded and found one or more roots. In this case, `tret` is the location of the root. If `nrtfn` >1, call *IDAGetRootInfo()* to see which g_i were found to have a root.
- `IDA_MEM_NULL` – The `ida_mem` argument was `NULL`.
- `IDA_ILL_INPUT` – One of the inputs to *IDASolve()* was illegal, or some other input to the solver was either illegal or missing. The latter category includes the following situations:
  - The tolerances have not been set.
  - A component of the error weight vector became zero during internal time-stepping.
  - The linear solver initialization function called by the user after calling *IDACreate()* failed to set the linear solver-specific `lsolve` field in `ida_mem`.
  - A root of one of the root functions was found both at a point t and also very near t.

  In any case, the user should see the printed error message for details.
- `IDA_TOO_MUCH_WORK` – The solver took `mxstep` internal steps but could not reach `tout`. The default value for `mxstep` is `MXSTEP_DEFAULT = 500`.
- `IDA_TOO_MUCH_ACC` – The solver could not satisfy the accuracy demanded by the user for some internal step.
- `IDA_ERR_FAIL` – Error test failures occurred too many times (`MXNEF = 10`) during one internal time step or occurred with $|h| = h_{\min}$.
- `IDA_CONV_FAIL` – Convergence test failures occurred too many times (`MXNCF = 10`) during one internal time step or occurred with $|h| = h_{\min}$.
- `IDA_LINIT_FAIL` – The linear solver's initialization function failed.
- `IDA_LSETUP_FAIL` – The linear solver's setup function failed in an unrecoverable manner.
- `IDA_LSOLVE_FAIL` – The linear solver's solve function failed in an unrecoverable manner.
- `IDA_CONSTR_FAIL` – The inequality constraints were violated and the solver was unable to recover.
- `IDA_REP_RES_ERR` – The user's residual function repeatedly returned a recoverable error flag, but the solver was unable to recover.
- `IDA_RES_FAIL` – The user's residual function returned a nonrecoverable error flag.
- `IDA_RTFUNC_FAIL` – The rootfinding function failed.

Notes: The vectors `yret` and `ypret` can occupy the same space as the initial condition vectors `y0` and `yp0`, respectively, that were passed to *IDAInit()*.

In the `IDA_ONE_STEP` mode, `tout` is used on the first call only, and only to get the direction and rough scale of the independent variable.

If a stop time is enabled (through a call to *IDASetStopTime()*), then *IDASolve()* returns the solution at `tstop`. Once the integrator returns at a stop time, any future testing for `tstop` is disabled (and can be reenabled only though a new call to *IDASetStopTime()*).

All failure return values are negative and therefore a test `flag < 0` will trap all *IDASolve()* failures.

On any error return in which one or more internal steps were taken by *IDASolve()*, the returned values of `tret`, `yret`, and `ypret` correspond to the farthest point reached in the integration. On all other error returns, these values are left unchanged from the previous *IDASolve()* return.

5.1.4.10 Optional input functions

There are numerous optional input parameters that control the behavior of the IDAS solver. IDAS provides functions that can be used to change these optional input parameters from their default values. The main inputs are divided in the following categories:

- Table 5.1 list the main IDAS optional inputs,
- Table 5.2 lists the IDALS linear solver interface optional inputs,
- Table 5.3 lists the IDANLS nonlinear solver interface optional inputs,
- Table 5.4 lists the initial condition calculation optional inputs, and
- Table 5.5 lists the rootfinding optional inputs.

These optional inputs are described in detail in the remainder of this section. For the most casual use of IDAS, the reader can skip to §5.1.5.

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

The optional input calls can, unless otherwise noted, be executed in any order. However, if the user's program calls either *IDASetErrFile()* or *IDASetErrHandlerFn()*, then that call should appear first, in order to take effect for any later error message. Finally, a call to an `IDASet***` function can, unless otherwise noted, be made at any time from the user's calling program and, if successful, takes effect immediately.

Main solver optional input functions

Table 5.1: Optional inputs for IDAS

Optional input	Function name	Default
Pointer to an error file	*IDASetErrFile()*	stderr
Error handler function	*IDASetErrHandlerFn()*	internal fn.
User data	*IDASetUserData()*	NULL
Maximum order for BDF method	*IDASetMaxOrd()*	5
Maximum no. of internal steps before t_{out}	*IDASetMaxNumSteps()*	500
Initial step size	*IDASetInitStep()*	estimated
Maximum absolute step size	*IDASetMaxStep()*	∞
Value of t_{stop}	*IDASetStopTime()*	∞
Maximum no. of error test failures	*IDASetMaxErrTestFails()*	10
Suppress alg. vars. from error test	*IDASetSuppressAlg()*	SUNFALSE
Variable types (differential/algebraic)	*IDASetId()*	NULL
Inequality constraints on solution	*IDASetConstraints()*	NULL

int **IDASetErrFile**(void *ida_mem, FILE *errfp)

The function *IDASetErrFile()* specifies the file pointer where all IDAS messages should be directed when using the default IDAS error handler function.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- errfp – pointer to output file.

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

Notes: The default value for errfp is stderr. Passing a value NULL disables all future error message output (except for the case in which the IDAS memory pointer is NULL). This use of *IDASetErrFile()* is strongly discouraged.

Warning: If *IDASetErrFile()* 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 **IDASetErrHandlerFn**(void *ida_mem, *IDAErrHandlerFn* ehfun, void *eh_data)

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

Arguments:

- ida_mem – pointer to the IDAS solver object.
- ehfun – is the user's error handler function. See *IDAErrHandlerFn* for more details.
- eh_data – pointer to user data passed to ehfun every time it is called.

Return value:

- IDA_SUCCESS – The function ehfun and data pointer eh_data have been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

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

int **IDASetUserData**(void *ida_mem, void *user_data)

The function *IDASetUserData()* attaches a user-defined data pointer to the main IDAS solver object.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `user_data` – pointer to the user data.

Return value:

- `IDA_SUCCESS` – The optional value has been successfully set.
- `IDA_MEM_NULL` – The `ida_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 *IDASetUserData()* must be made before the call to specify the linear solver.

int **IDASetMaxOrd**(void *ida_mem, int maxord)

The function *IDASetMaxOrd()* specifies the maximum order of the linear multistep method.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `maxord` – value of the maximum method order. This must be positive.

Return value:

- `IDA_SUCCESS` – The optional value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDA_ILL_INPUT` – The input value `maxord` is ≤ 0 , or larger than the max order value when *IDAInit()* was called.

Notes: The default value is 5. If the input value exceeds 5, the value 5 will be used. If called before *IDAInit()*, `maxord` limits the memory requirements for the internal IDAS memory block and its value cannot be increased past the value set when *IDAInit()* was called.

int **IDASetMaxNumSteps**(void *ida_mem, long int mxsteps)

The function *IDASetMaxNumSteps()* specifies the maximum number of steps to be taken by the solver in its attempt to reach the next output time.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `mxsteps` – maximum allowed number of steps.

Return value:

- `IDA_SUCCESS` – The optional value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.

Notes: Passing `mxsteps` = 0 results in IDAS using the default value (500). Passing `mxsteps` < 0 disables the test (not recommended).

int **IDASetInitStep**(void *ida_mem, *realtype* hin)
The function *IDASetInitStep()* specifies the initial step size.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `hin` – value of the initial step size to be attempted. Pass 0.0 to have IDAS use the default value.

Return value:

- `IDA_SUCCESS` – The optional value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.

Notes: By default, IDAS estimates the initial step as the solution of $\|h\dot{y}\|_{WRMS} = 1/2$, with an added restriction that $|h| \leq .001|t_{\text{out}} - t_0|$.

int **IDASetMaxStep**(void *ida_mem, *realtype* hmax)
The function *IDASetMaxStep()* specifies the maximum absolute value of the step size.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `hmax` – maximum absolute value of the step size.

Return value:

- `IDA_SUCCESS` – The optional value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDA_ILL_INPUT` – Either `hmax` is not positive or it is smaller than the minimum allowable step.

Notes: Pass `hmax = 0` to obtain the default value ∞.

int **IDASetStopTime**(void *ida_mem, *realtype* tstop)
The function *IDASetStopTime()* specifies the value of the independent variable t past which the solution is not to proceed.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `tstop` – value of the independent variable past which the solution should not proceed.

Return value:

- `IDA_SUCCESS` – The optional value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDA_ILL_INPUT` – The value of `tstop` is not beyond the current t value, t_n.

Notes: The default, if this routine is not called, is that no stop time is imposed. Once the integrator returns at a stop time, any future testing for `tstop` is disabled (and can be reenabled only though a new call to *IDASetStopTime()*).

int **IDASetMaxErrTestFails**(void *ida_mem, int maxnef)
The function *IDASetMaxErrTestFails()* specifies the maximum number of error test failures in attempting one step.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `maxnef` – maximum number of error test failures allowed on one step (>0).

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

Notes: The default value is 10.

int **IDASetSuppressAlg**(void *ida_mem, *booleantype* suppressalg)
The function *IDASetSuppressAlg()* indicates whether or not to suppress algebraic variables in the local error test.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- suppressalg – indicates whether to suppress (SUNTRUE) or include (SUNFALSE) the algebraic variables in the local error test.

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

Notes: The default value is SUNFALSE. If suppressalg = SUNTRUE is selected, then the id vector must be set (through *IDASetId()*) to specify the algebraic components. In general, the use of this option (with suppressalg = SUNTRUE) is *discouraged* when solving DAE systems of index 1, whereas it is generally *encouraged* for systems of index 2 or more. See pp. 146-147 of [4] for more on this issue.

int **IDASetId**(void *ida_mem, *N_Vector* id)
The function *IDASetId()* specifies algebraic/differential components in the y vector.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- id – a vector of values identifying the components of y as differential or algebraic variables. A value of 1.0 indicates a differential variable, while 0.0 indicates an algebraic variable.

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

Notes: The vector id is required if the algebraic variables are to be suppressed from the local error test (see *IDASetSuppressAlg()*) or if *IDACalcIC()* is to be called with icopt = IDA_YA_YDP_INIT.

int **IDASetConstraints**(void *ida_mem, *N_Vector* constraints)
The function *IDASetConstraints()* specifies a vector defining inequality constraints for each component of the solution vector y.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- constraints – vector of constraint flags.
  - If constraints[i] = 0, no constraint is imposed on y_i.
  - If constraints[i] = 1, y_i will be constrained to be $y_i \geq 0.0$.
  - If constraints[i] = -1, y_i will be constrained to be $y_i \leq 0.0$.
  - If constraints[i] = 2, y_i will be constrained to be $y_i > 0.0$.
  - If constraints[i] = -2, y_i will be constrained to be $y_i < 0.0$.

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_ILL_INPUT – The constraints vector contains illegal values or the simultaneous corrector option has been selected when doing forward sensitivity analysis.

Notes: The presence of a non-NULL constraints vector that is not 0.0 in all components will cause constraint checking to be performed. However, a call with 0.0 in all components of constraints vector will result in an illegal input return. A NULL input will disable constraint checking.

Constraint checking when doing forward sensitivity analysis with the simultaneous corrector option is currently disallowed and will result in an illegal input return.

Linear solver interface optional input functions

Table 5.2: Optional inputs for the IDALS linear solver interface

Optional input	Function name	Default
Jacobian function	*IDASetJacFn()*	DQ
Enable or disable linear solution scaling	*IDASetLinearSolutionScaling()*	on
Jacobian-times-vector function	*IDASetJacTimes()*	NULL, DQ
Preconditioner functions	*IDASetPreconditioner()*	NULL, NULL
Ratio between linear and nonlinear tolerances	*IDASetEpsLin()*	0.05
Increment factor used in DQ Jv approx.	*IDASetIncrementFactor()*	1.0
Jacobian-times-vector DQ Res function	*IDASetJacTimesResFn()*	NULL
Newton linear solve tolerance conversion factor	*IDASetLSNormFactor()*	vector length

The mathematical explanation of the linear solver methods available to IDAS is provided in §2.1. We group the user-callable routines into four categories: general routines concerning the overall IDALS linear solver interface, optional inputs for matrix-based linear solvers, optional inputs for matrix-free linear solvers, and optional inputs for iterative linear solvers. We note that the matrix-based and matrix-free groups are mutually exclusive, whereas the "iterative" tag can apply to either case.

When using matrix-based linear solver modules, the IDALS solver interface needs a function to compute an approximation to the Jacobian matrix $J(t, y, \dot{y})$. This function must be of type *IDALsJacFn*. The user can supply a Jacobian function or, if using the *SUNMATRIX_DENSE* or *SUNMATRIX_BAND* modules for the matrix J, can use the default internal difference quotient approximation that comes with the IDALS interface. To specify a user-supplied Jacobian function jac, IDALS provides the function *IDASetJacFn()*. The IDALS 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 *IDASetUserData()*.

int **IDASetJacFn**(void *ida_mem, *IDALsJacFn* jac)

The function *IDASetJacFn()* specifies the Jacobian approximation function to be used for a matrix-based solver within the IDALS interface.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- jac – user-defined Jacobian approximation function. See *IDALsJacFn* for more details.

Return value:

- IDALS_SUCCESS – The optional value has been successfully set.

- IDALS_MEM_NULL – The ida_mem pointer is NULL.
- IDALS_LMEM_NULL – The IDALS linear solver interface has not been initialized.

Notes: This function must be called after the IDALS linear solver interface has been initialized through a call to *IDASetLinearSolver()*. By default, IDALS 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 IDADlsSetJacFn() 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 a matrix-based linear solver the matrix information will be updated infrequently to reduce matrix construction and, with direct solvers, factorization costs. As a result the value of α may not be current and a scaling factor is applied to the solution of the linear system to account for the lagged value of α. See §8.2.1 for more details. The function *IDASetLinearSolutionScaling()* can be used to disable this scaling when necessary, e.g., when providing a custom linear solver that updates the matrix using the current α as part of the solve.

int **IDASetLinearSolutionScaling**(void *ida_mem, *booleantype* onoff)

The function *IDASetLinearSolutionScaling()* enables or disables scaling the linear system solution to account for a change in α in the linear system. For more details see §8.2.1.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- onoff – flag to enable (SUNTRUE) or disable (SUNFALSE) scaling.

Return value:

- IDALS_SUCCESS – The flag value has been successfully set.
- IDALS_MEM_NULL – The ida_mem pointer is NULL.
- IDALS_LMEM_NULL – The IDALS linear solver interface has not been initialized.
- IDALS_ILL_INPUT – The attached linear solver is not matrix-based.

Notes: This function must be called after the IDALS linear solver interface has been initialized through a call to *IDASetLinearSolver()*. By default scaling is enabled with matrix-based linear solvers.

When using matrix-free linear solver modules, the IDALS solver interface requires a function to compute an approximation to the product between the Jacobian matrix $J(t, y, \dot{y})$ and a vector v. The user can supply a Jacobian-times-vector approximation function, or use the default internal difference quotient function that comes with the IDALS solver interface.

A user-defined Jacobian-vector product function must be of type *IDALsJacTimesVecFn* and can be specified through a call to *IDASetJacTimes()*. The evaluation and processing of any Jacobian-related data needed by the user's Jacobian-vector product function may be done in the optional user-supplied function jtsetup (see §5.1.5.7 for specification details). The pointer user_data received through *IDASetUserData()* (or a pointer to NULL if user_data was not specified) is passed to the Jacobian-vector product setup and product functions, jtsetup and jtimes, each time they are 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 **IDASetJacTimes**(void *ida_mem, *IDALsJacTimesSetupFn* jsetup, *IDALsJacTimesVecFn* jtimes)

The function *IDASetJacTimes()* specifies the Jacobian-vector product setup and product functions.

Arguments:

- ida_mem – pointer to the IDAS solver object.

- `jtsetup` – user-defined function to set up the Jacobian-vector product. See *IDALsJacTimesSetupFn* for more details. Pass `NULL` if no setup is necessary.
- `jtimes` – user-defined Jacobian-vector product function. See *IDALsJacTimesVecFn* for more details.

Return value:

- `IDALS_SUCCESS` – The optional value has been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.
- `IDALS_SUNLS_FAIL` – An error occurred when setting up the system matrix-times-vector routines in the `SUNLinearSolver` object used by the IDALS interface.

Notes: The default is to use an internal finite difference quotient for `jtimes` and to omit `jtsetup`. If `NULL` is passed to `jtimes`, these defaults are used. A user may specify non-`NULL` `jtimes` and `NULL` `jtsetup` inputs. This function must be called after the IDALS linear solver interface has been initialized through a call to *IDASetLinearSolver()*.

Warning: The previous routine `IDASpilsSetJacTimes()` 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 default difference-quotient approximation to the Jacobian-vector product, the user may specify the factor to use in setting increments for the finite-difference approximation, via a call to *IDASetIncrementFactor()*.

int **IDASetIncrementFactor**(void *ida_mem, *realtype* dqincfac)

The function *IDASetIncrementFactor()* specifies the increment factor to be used in the difference-quotient approximation to the product Jv. Specifically, Jv is approximated via the formula

$$Jv = \frac{1}{\sigma}\left[F(t, \tilde{y}, \tilde{\dot{y}}) - F(t, y, \dot{y})\right],$$

where $\tilde{y} = y + \sigma v$, $\tilde{\dot{y}} = \dot{y} + c_j \sigma v$, c_j is a BDF parameter proportional to the step size, $\sigma =$ `dqincfac`$\sqrt{N}$, and N is the number of equations in the DAE system.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `dqincfac` – user-specified increment factor positive.

Return value:

- `IDALS_SUCCESS` – The optional value has been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.
- `IDALS_ILL_INPUT` – The specified value of `dqincfac` is ≤ 0.

Notes: The default value is 1.0. This function must be called after the IDALS linear solver interface has been initialized through a call to *IDASetLinearSolver()*.

Warning: The previous routine `IDASpilsSetIncrementFactor()` 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.

Additionally, when using the internal difference quotient, the user may also optionally supply an alternative residual function for use in the Jacobian-vector product approximation by calling *IDASetJacTimesResFn()*. The alternative residual function should compute a suitable (and differentiable) approximation to the residual function provided to *IDAInit()*. For example, as done in [22] for an ODE in explicit form, the alternative function may use lagged values when evaluating a nonlinearity to avoid differencing a potentially non-differentiable factor.

int **IDASetJacTimesResFn**(void *ida_mem, *IDAResFn* jtimesResFn)
The function *IDASetJacTimesResFn()* specifies an alternative DAE residual function for use in the internal Jacobian-vector product difference quotient approximation.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `jtimesResFn` – is the function which computes the alternative DAE residual function to use in Jacobian-vector product difference quotient approximations. See *IDAResFn* for more details.

Return value:

- `IDALS_SUCCESS` – The optional value has been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.
- `IDALS_ILL_INPUT` – The internal difference quotient approximation is disabled.

Notes: The default is to use the residual function provided to *IDAInit()* in the internal difference quotient. If the input resudual function is `NULL`, the default is used. This function must be called after the IDALS linear solver interface has been initialized through a call to *IDASetLinearSolver()*.

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 IDAS using the function *IDASetPreconditioner()*. 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 §5.1.5.8 and §5.1.5.9). The user data pointer received through *IDASetUserData()* (or `NULL` if a user data pointer 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 **IDASetPreconditioner**(void *ida_mem, *IDALsPrecSetupFn* psetup, *IDALsPrecSolveFn* psolve)
The function *IDASetPreconditioner()* specifies the preconditioner setup and solve functions.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `psetup` – user-defined function to set up the preconditioner. See *IDALsPrecSetupFn* for more details. Pass `NULL` if no setup is necessary.
- `psolve` – user-defined preconditioner solve function. See *IDALsPrecSolveFn* for more details.

Return value:

- `IDALS_SUCCESS` – The optional values have been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.
- `IDALS_SUNLS_FAIL` – An error occurred when setting up preconditioning in the `SUNLinearSolver` object used by the IDALS interface.

Notes: The default is `NULL` for both arguments (i.e., no preconditioning). This function must be called after the IDALS linear solver interface has been initialized through a call to *IDASetLinearSolver()*.

Warning: The previous routine `IDASpilsSetPreconditioner()` 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.

Also, as described in §2.1, the IDALS interface requires that iterative linear solvers stop when the norm of the preconditioned residual satisfies

$$\|r\| \leq \frac{\epsilon_L \epsilon}{10}$$

where ϵ is the nonlinear solver tolerance, and the default $\epsilon_L = 0.05$; this value may be modified by the user through the *IDASetEpsLin()* function.

int **IDASetEpsLin**(void *ida_mem, *realtype* eplifac)

The function *IDASetEpsLin()* specifies the factor by which the Krylov linear solver's convergence test constant is reduced from the nonlinear iteration test constant.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `eplifac` – linear convergence safety factor ≥ 0.0.

Return value:

- `IDALS_SUCCESS` – The optional value has been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.
- `IDALS_ILL_INPUT` – The factor `eplifac` is negative.

Notes: The default value is 0.05. This function must be called after the IDALS linear solver interface has been initialized through a call to *IDASetLinearSolver()*. If `eplifac` $= 0.0$ is passed, the default value is used.

Warning: The previous routine `IDASpilsSetEpsLin()` 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.

int **IDASetLSNormFactor**(void *ida_mem, *realtype* nrmfac)

The function *IDASetLSNormFactor()* specifies the factor to use when converting from the integrator tolerance (WRMS norm) to the linear solver tolerance (L2 norm) for Newton linear system solves e.g., `tol_L2 = fac * tol_WRMS`.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `nrmfac` – the norm conversion factor.
  - If `nrmfac > 0`, the provided value is used.
  - If `nrmfac = 0` then the conversion factor is computed using the vector length i.e., `nrmfac = N_VGetLength(y)` (*default*).
  - If `nrmfac < 0` then the conversion factor is computed using the vector dot product `nrmfac = N_VDotProd(v,v)` where all the entries of `v` are one.

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

Notes: This function must be called after the IDALS linear solver interface has been initialized through a call to *IDASetLinearSolver()*. Prior to the introduction of *N_VGetLength()* in SUNDIALS v5.0.0 (IDAS v4.0.0) the value of nrmfac was computed using *N_VDotProd()* i.e., the nrmfac < 0 case.

Nonlinear solver interface optional input functions

Table 5.3: Optional inputs for the IDANLS nonlinear solver interface

Optional input	Function name	Default
Maximum no. of nonlinear iterations	*IDASetMaxNonlinIters()*	4
Maximum no. of convergence failures	*IDASetMaxConvFails()*	10
Coeff. in the nonlinear convergence test	*IDASetNonlinConvCoef()*	0.33
Residual function for nonlinear system evaluations	*IDASetNlsResFn()*	NULL

The following functions can be called to set optional inputs controlling the nonlinear solver.

int **IDASetMaxNonlinIters**(void *ida_mem, int maxcor)

The function *IDASetMaxNonlinIters()* specifies the maximum number of nonlinear solver iterations in one solve attempt.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- maxcor – maximum number of nonlinear solver iterations allowed in one solve attempt (>0).

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_MEM_FAIL – The SUNNonlinearSolver object is NULL.

Notes: The default value is 4.

int **IDASetMaxConvFails**(void *ida_mem, int maxncf)

The function *IDASetMaxConvFails()* specifies the maximum number of nonlinear solver convergence failures in one step.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- maxncf – maximum number of allowable nonlinear solver convergence failures in one step (>0).

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

Notes: The default value is 10.

int **IDASetNonlinConvCoef**(void *ida_mem, *realtype* nlscoef)

The function *IDASetNonlinConvCoef()* specifies the safety factor in the nonlinear convergence test; see (2.8).

Arguments:

- ida_mem – pointer to the IDAS solver object.

- `nlscoef` – coefficient in nonlinear convergence test (>0.0).

Return value:

- `IDA_SUCCESS` – The optional value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDA_ILL_INPUT` – The value of `nlscoef` is ≤ 0.0.

Notes: The default value is 0.33.

int **IDASetNlsResFn**(void *ida_mem, *IDAResFn* res)

The function *IDASetNlsResFn()* specifies an alternative residual function for use in nonlinear system function evaluations.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `res` – the alternative function which computes the DAE residual function $F(t, y, \dot{y})$. See *IDAResFn* for more details.

Return value:

- `IDA_SUCCESS` – The optional function has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.

Notes: The default is to use the residual function provided to *IDAInit()* in nonlinear system function evaluations. If the input residual function is `NULL`, the default is used.

When using a non-default nonlinear solver, this function must be called after *IDASetNonlinearSolver()*.

When doing forward sensitivity analysis with the simultaneous solver strategy is function must be called after *IDASetNonlinearSolverSensSim()*.

Initial condition calculation optional input functions

Table 5.4: Optional inputs for IDAS initial condition calculation

Optional input	Function name	Default
Coeff. in the nonlinear convergence test	*IDASetNonlinConvCoefIC()*	0.0033
Maximum no. of steps	*IDASetMaxNumStepsIC()*	5
Maximum no. of Jacobian/precond. evals.	*IDASetMaxNumJacsIC()*	4
Maximum no. of Newton iterations	*IDASetMaxNumItersIC()*	10
Max. linesearch backtracks per Newton iter.	*IDASetMaxBacksIC()*	100
Turn off linesearch	*IDASetLineSearchOffIC()*	SUNFALSE
Lower bound on Newton step	*IDASetStepToleranceIC()*	uround$^{2/3}$

The following functions can be called just prior to calling *IDACalcIC()* to set optional inputs controlling the initial condition calculation.

int **IDASetNonlinConvCoefIC**(void *ida_mem, *realtype* epiccon)

The function *IDASetNonlinConvCoefIC()* specifies the positive constant in the Newton iteration convergence test within the initial condition calculation.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `epiccon` – coefficient in the Newton convergence test (> 0).

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_ILL_INPUT – The epiccon factor is ≤ 0.0.

Notes: The default value is $0.01 \cdot 0.33$. This test uses a weighted RMS norm (with weights defined by the tolerances). For new initial value vectors y and $\dot{y}$ to be accepted, the norm of $J^{-1}F(t_0, y, \dot{y})$ must be $\leq$ epiccon, where J is the system Jacobian.

int **IDASetMaxNumStepsIC**(void *ida_mem, int maxnh)

The function *IDASetMaxNumStepsIC()* specifies the maximum number of steps allowed when icopt = IDA_YA_YDP_INIT in *IDACalcIC()*, where h appears in the system Jacobian, $J = \dfrac{\partial F}{\partial y} + \left(\dfrac{1}{h}\right)\dfrac{\partial F}{\partial \dot{y}}$.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- maxnh – maximum allowed number of values for h.

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_ILL_INPUT – maxnh is non-positive.

Notes: The default value is 5.

int **IDASetMaxNumJacsIC**(void *ida_mem, int maxnj)

The function *IDASetMaxNumJacsIC()* specifies the maximum number of the approximate Jacobian or preconditioner evaluations allowed when the Newton iteration appears to be slowly converging.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- maxnj – maximum allowed number of Jacobian or preconditioner evaluations.

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_ILL_INPUT – maxnj is non-positive.

Notes: The default value is 4.

int **IDASetMaxNumItersIC**(void *ida_mem, int maxnit)

The function *IDASetMaxNumItersIC()* specifies the maximum number of Newton iterations allowed in any one attempt to solve the initial conditions calculation problem.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- maxnit – maximum number of Newton iterations.

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

- IDA_ILL_INPUT – maxnit is non-positive.

Notes: The default value is 10.

int **IDASetMaxBacksIC**(void *ida_mem, int maxbacks)

The function *IDASetMaxBacksIC()* specifies the maximum number of linesearch backtracks allowed in any Newton iteration, when solving the initial conditions calculation problem.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- maxbacks – maximum number of linesearch backtracks per Newton step.

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_ILL_INPUT – maxbacks is non-positive.

Notes: The default value is 100.

If *IDASetMaxBacksIC()* is called in a Forward Sensitivity Analysis, the the limit maxbacks applies in the calculation of both the initial state values and the initial sensititivies.

int **IDASetLineSearchOffIC**(void *ida_mem, *booleantype* lsoff)

The function *IDASetLineSearchOffIC()* specifies whether to turn on or off the linesearch algorithm.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- lsoff – a flag to turn off (SUNTRUE) or keep (SUNFALSE) the linesearch algorithm.

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

Notes:

The default value is SUNFALSE.

int **IDASetStepToleranceIC**(void *ida_mem, int steptol)

The function *IDASetStepToleranceIC()* specifies a positive lower bound on the Newton step.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- steptol – Minimum allowed WRMS-norm of the Newton step (> 0.0).

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_ILL_INPUT – The steptol tolerance is ≤ 0.0.

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

Rootfinding optional input functions

Table 5.5: Optional inputs for IDAS rootfinding

Optional input	Function name	Default
Direction of zero-crossing	*IDASetRootDirection()*	both
Disable rootfinding warnings	*IDASetNoInactiveRootWarn()*	none

The following functions can be called to set optional inputs to control the rootfinding algorithm.

int **IDASetRootDirection**(void *ida_mem, int *rootdir)

The function *IDASetRootDirection()* specifies the direction of zero-crossings to be located and returned to the user.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `rootdir` – state array of length `nrtfn` , the number of root functions g_i , as specified in the call to the function *IDARootInit()*.
  - A value of 0 for `rootdir[i]` indicates that crossing in either direction should be reported for g_i.
  - A value of $+1$ or -1 for `rootdir[i]` indicates that the solver should report only zero-crossings where g_i is increasing or decreasing, respectively.

Return value:

- `IDA_SUCCESS` – The optional value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDA_ILL_INPUT` – rootfinding has not been activated through a call to *IDARootInit()*.

Notes: The default behavior is to locate both zero-crossing directions.

int **IDASetNoInactiveRootWarn**(void *ida_mem)

The function *IDASetNoInactiveRootWarn()* disables issuing a warning if some root function appears to be identically zero at the beginning of the integration.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.

Return value:

- `IDA_SUCCESS` – The optional value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.

Notes: IDAS will not report the initial conditions as a possible zero-crossing (assuming that one or more components g_i are zero at the initial time). However, if it appears that some g_i is identically zero at the initial time (i.e., g_i is zero at the initial time and after the first step), IDAS will issue a warning which can be disabled with this optional input function.

5.1.4.11 Interpolated output function

An optional function *IDAGetDky()* is available to obtain additional output values. This function must be called after a successful return from *IDASolve()* and provides interpolated values of y or its derivatives of order up to the last internal order used for any value of t in the last internal step taken by IDAS.

int **IDAGetDky**(void *ida_mem, *realtype* t, int k, *N_Vector* dky)

The function *IDAGetDky()* computes the interpolated values of the k^{th} derivative of y for any value of t in the last internal step taken by IDAS. The value of k must be non-negative and smaller than the last internal order used. A value of 0 for k means that the y is interpolated. The value of t must satisfy $t_n - h_u \leq t \leq t_n$, where t_n denotes the current internal time reached, and h_u is the last internal step size used successfully.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `t` – time at which to interpolate.
- `k` – integer specifying the order of the derivative of y wanted.
- `dky` – vector containing the interpolated k^{th} derivative of $y(t)$.

Return value:

- `IDA_SUCCESS` – *IDAGetDky()* succeeded.
- `IDA_MEM_NULL` – The `ida_mem` argument was `NULL`.
- `IDA_BAD_T` – `t` is not in the interval $[t_n - h_u, t_n]$.
- `IDA_BAD_K` – `k` is not one of $0, 1, \ldots, k_{\text{last}}$.
- `IDA_BAD_DKY` – `dky` is `NULL`.

Notes: It is only legal to call the function *IDAGetDky()* after a successful return from *IDASolve()*. Functions *IDAGetCurrentTime()*, *IDAGetLastStep()* and *IDAGetLastOrder()* can be used to access t_n, h_u, and k_{last}.

5.1.4.12 Optional output functions

IDAS provides an extensive list of functions that can be used to obtain solver performance information. Table 5.6 lists all optional output functions in IDAS, which are then described in detail in the remainder of this section.

Some of the optional outputs, especially the various counters, can be very useful in determining how successful the IDAS solver is in doing its job. For example, the counters `nsteps` and `nrevals` provide a rough measure of the overall cost of a given run, and can be compared among runs with differing input options to suggest which set of options is most efficient. The ratio `nniters/nsteps` measures the performance of the nonlinear solver in solving the nonlinear systems at each time step; typical values for this range from 1.1 to 1.8. The ratio `njevals/nniters` (in the case of a matrix-based linear solver), and the ratio `npevals/nniters` (in the case of an iterative linear solver) measure the overall degree of nonlinearity in these systems, and also the quality of the approximate Jacobian or preconditioner being used. Thus, for example, `njevals/nniters` can indicate if a user-supplied Jacobian is inaccurate, if this ratio is larger than for the case of the corresponding internal Jacobian. The ratio `nliters/nniters` measures the performance of the Krylov iterative linear solver, and thus (indirectly) the quality of the preconditioner.

Table 5.6: Optional outputs for IDAS, IDALS, and IDANLS

Optional output	Function name
Size of IDAS real and integer workspace	*IDAGetWorkSpace()*
Cumulative number of internal steps	*IDAGetNumSteps()*

continues on next page

Table 5.6 – continued from previous page

Optional output	Function name
No. of calls to residual function	*IDAGetNumResEvals()*
No. of calls to linear solver setup function	*IDAGetNumLinSolvSetups()*
No. of local error test failures that have occurred	*IDAGetNumErrTestFails()*
Order used during the last step	*IDAGetLastOrder()*
Order to be attempted on the next step	*IDAGetCurrentOrder()*
Actual initial step size used	*IDAGetActualInitStep()*
Step size used for the last step	*IDAGetLastStep()*
Step size to be attempted on the next step	*IDAGetCurrentStep()*
Current internal time reached by the solver	*IDAGetCurrentTime()*
Suggested factor for tolerance scaling	*IDAGetTolScaleFactor()*
Error weight vector for state variables	*IDAGetErrWeights()*
Estimated local errors	*IDAGetEstLocalErrors()*
No. of nonlinear solver iterations	*IDAGetNumNonlinSolvIters()*
No. of nonlinear convergence failures	*IDAGetNumNonlinSolvConvFails()*
Array showing roots found	*IDAGetRootInfo()*
No. of calls to user root function	*IDAGetNumGEvals()*
Name of constant associated with a return flag	*IDAGetReturnFlagName()*
Number of backtrack operations	*IDAGetNumBacktrackOps()*
Corrected initial conditions	*IDAGetConsistentIC()*
Size of real and integer workspace	*IDAGetLinWorkSpace()*
No. of Jacobian evaluations	*IDAGetNumJacEvals()*
No. of residual calls for finite diff. Jacobian-vector evals.	*IDAGetNumLinResEvals()*
No. of linear iterations	*IDAGetNumLinIters()*
No. of linear convergence failures	*IDAGetNumLinConvFails()*
No. of preconditioner evaluations	*IDAGetNumPrecEvals()*
No. of preconditioner solves	*IDAGetNumPrecSolves()*
No. of Jacobian-vector setup evaluations	*IDAGetNumJTSetupEvals()*
No. of Jacobian-vector product evaluations	*IDAGetNumJtimesEvals()*
Last return from a linear solver function	*IDAGetLastLinFlag()*
Name of constant associated with a return flag	*IDAGetLinReturnFlagName()*

Main solver optional output functions

IDAS 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, solver performance statistics, as well as additional data from the IDAS solver object (a suggested tolerance scaling factor, the error weight vector, and the vector of estimated local errors). Also provided are functions to extract statistics related to the performance of the nonlinear solver being used. As a convenience, additional extraction functions provide the optional outputs in groups. These optional output functions are described next.

int **IDAGetWorkSpace**(void *ida_mem, long int *lenrw, long int *leniw)

The function *IDAGetWorkSpace()* returns the IDAS real and integer workspace sizes.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `lenrw` – number of real values in the IDAS workspace.
- `leniw` – number of integer values in the IDAS workspace.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

Notes: In terms of the problem size N, the maximum method order maxord, and the number of root functions nrtfn (see §5.1.4.8), the actual size of the real workspace, in *realtype* words, is given by the following:

- base value: lenrw $= 55 + (m + 6) * N_r + 3 *$ nrtfn;
- with *IDASVtolerances()*: lenrw $=$ lenrw $+ N_r$;
- with constraint checking (see *IDASetConstraints()*): lenrw $=$ lenrw $+ N_r$;
- with id specified (see *IDASetId()*): lenrw $=$ lenrw $+ N_r$;

where $m = \max($maxord$, 3)$, and N_r is the number of real words in one N_Vector ($\approx N$).

The size of the integer workspace (without distinction between int and long int words) is given by:

- base value: leniw $= 38 + (m + 6) * N_i +$ nrtfn;
- with *IDASVtolerances()*: leniw $=$ leniw $+ N_i$;
- with constraint checking: lenrw $=$ lenrw $+ N_i$;
- with id specified (see *IDASetId()*): lenrw $=$ lenrw $+ N_i$;

where N_i is the number of integer words in one N_Vector (= 1 for the serial N_Vector and 2 * npes for the parallel N_Vector on npes processors). For the default value of maxord, with no rootfinding, no id, no constraints, and with no call to *IDASVtolerances()*, these lengths are given roughly by lenrw $= 55 + 11 * N$ and leniw $= 49$.

Note that additional memory is allocated if quadratures and/or forward sensitivity integration is enabled. See §5.2.1 and §5.4.2.1 for more details.

int **IDAGetNumSteps**(void *ida_mem, long int *nsteps)

The function *IDAGetNumSteps()* returns the cumulative number of internal steps taken by the solver (total so far).

Arguments:

- ida_mem – pointer to the IDAS solver object.
- nsteps – number of steps taken by IDAS.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

int **IDAGetNumResEvals**(void *ida_mem, long int *nrevals)

The function *IDAGetNumResEvals()* returns the number of calls to the user's residual evaluation function.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- nrevals – number of calls to the user's res function.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

Notes: The nrevals value returned by *IDAGetNumResEvals()* does not account for calls made to res from a linear solver or preconditioner module.

int **IDAGetNumLinSolvSetups**(void *ida_mem, long int *nlinsetups)
The function *IDAGetNumLinSolvSetups()* returns the cumulative number of calls made to the linear solver's setup function (total so far).

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `nlinsetups` – number of calls made to the linear solver setup function.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.

int **IDAGetNumErrTestFails**(void *ida_mem, long int *netfails)
The function *IDAGetNumErrTestFails()* returns the cumulative number of local error test failures that have occurred (total so far).

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `netfails` – number of error test failures.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.

int **IDAGetLastOrder**(void *ida_mem, int *klast)
The function *IDAGetLastOrder()* returns the integration method order used during the last internal step.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `klast` – method order used on the last internal step.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.

int **IDAGetCurrentOrder**(void *ida_mem, int *kcur)
The function *IDAGetCurrentOrder()* returns the integration method order to be used on the next internal step.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `kcur` – method order to be used on the next internal step.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.

int **IDAGetLastStep**(void *ida_mem, *realtype* *hlast)
The function *IDAGetLastStep()* returns the integration step size taken on the last internal step (if from *IDASolve()*), or the last value of the artificial step size h (if from *IDACalcIC()*).

Arguments:

- `ida_mem` – pointer to the IDAS solver object.

- `hlast` – step size taken on the last internal step by IDAS, or last artificial step size used in *IDACalcIC()* , whichever was called last.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.

int **IDAGetCurrentStep**(void *ida_mem, *realtype* *hcur)
The function *IDAGetCurrentStep()* returns the integration step size to be attempted on the next internal step.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `hcur` – step size to be attempted on the next internal step.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.

int **IDAGetActualInitStep**(void *ida_mem, *realtype* *hinused)
The function *IDAGetActualInitStep()* returns the value of the integration step size used on the first step.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `hinused` – actual value of initial step size.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.

Notes:

Even if the value of the initial integration step size was specified by the user through a call to *IDASetInitStep()*, this value might have been changed by IDAS to ensure that the step size is within the prescribed bounds ($h_{min} \leq h_0 \leq h_{max}$), or to meet the local error test.

int **IDAGetCurrentTime**(void *ida_mem, *realtype* *tcur)
The function *IDAGetCurrentTime()* returns the current internal time reached by the solver.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `tcur` – current internal time reached.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.

int **IDAGetTolScaleFactor**(void *ida_mem, *realtype* *tolsfac)
The function *IDAGetTolScaleFactor()* returns a suggested factor by which the user's tolerances should be scaled when too much accuracy has been requested for some internal step.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `tolsfac` – suggested scaling factor for user tolerances.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

int **IDAGetErrWeights**(void *ida_mem, *N_Vector* eweight)

The function *IDAGetErrWeights()* returns the solution error weights at the current time. These are the W_i given by (2.7) (or by the user's *IDAEwtFn*).

Arguments:

- ida_mem – pointer to the IDAS solver object.
- eweight – solution error weights at the current time.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

Warning: The user must allocate space for eweight.

int **IDAGetEstLocalErrors**(void *ida_mem, *N_Vector* ele)

The function *IDAGetEstLocalErrors()* returns the estimated local errors.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- ele – estimated local errors at the current time.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

Warning: The user must allocate space for ele. The values returned in ele are only valid if *IDASolve()* returned a non-negative value.

Note: The ele vector, togther with the eweight vector from *IDAGetErrWeights()*, can be used to determine how the various components of the system contributed to the estimated local error test. Specifically, that error test uses the RMS norm of a vector whose components are the products of the components of these two vectors. Thus, for example, if there were recent error test failures, the components causing the failures are those with largest values for the products, denoted loosely as eweight[i]*ele[i].

---

int **IDAGetIntegratorStats**(void *ida_mem, long int *nsteps, long int *nrevals, long int *nlinsetups, long int *netfails, int *klast, int *kcur, *realtype* *hinused, *realtype* *hlast, *realtype* *hcur, *realtype* *tcur)

The function *IDAGetIntegratorStats()* returns the IDAS integrator stats in one function call.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- nsteps – cumulative number of steps taken by IDAS.

- nrevals – cumulative number of calls to the user's res functions.
- nlinsetups – cumulative number of calls made to the linear solver setup function.
- netfails – cumulative number of error test failures.
- klast – method order used on the last internal step.
- kcur – method order to be used on the next internal step.
- hinused – actual value of initial step size.
- hlast – step sized taken on the last internal step.
- hcur – step size to be attempted on the next internal step.
- tcur – current internal time reached.

Return value:

- IDA_SUCCESS – The optional output values have been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

int **IDAGetNumNonlinSolvIters**(void *ida_mem, long int *nniters)
The function *IDAGetNumNonlinSolvIters()* returns the cumulative number of nonlinear iterations performed.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- nniters – number of nonlinear iterations performed.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_MEM_FAIL – The SUNNonlinearSolver object is NULL.

int **IDAGetNumNonlinSolvConvFails**(void *ida_mem, long int *nncfails)
The function *IDAGetNumNonlinSolvConvFails()* returns the cumulative number of nonlinear convergence failures that have occurred.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- nncfails – number of nonlinear convergence failures.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

int **IDAGetNonlinSolvStats**(void *ida_mem, long int *nniters, long int *nncfails)
The function *IDAGetNonlinSolvStats()* returns the IDAS nonlinear solver statistics as a group.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- nniters – cumulative number of nonlinear iterations performed.
- nncfails – cumulative number of nonlinear convergence failures.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_MEM_FAIL – The SUNNonlinearSolver object is NULL.

char ***IDAGetReturnFlagName**(long int flag)
The function *IDAGetReturnFlagName()* returns the name of the IDAS constant corresponding to flag.

Arguments:

- flag – the flag returned by a call to an IDAS function.

Return value:

- char* – the flag name string.

Initial condition calculation optional output functions

int **IDAGetNumBacktrackOps**(void *ida_mem, long int *nbacktr)
The function *IDAGetNumBacktrackOps()* returns the number of backtrack operations done in the linesearch algorithm in *IDACalcIC()*.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- nbacktr – the cumulative number of backtrack operations.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

int **IDAGetConsistentIC**(void *ida_mem, *N_Vector* yy0_mod, *N_Vector* yp0_mod)
The function *IDAGetConsistentIC()* returns the corrected initial conditions calculated by *IDACalcIC()*.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- yy0_mod – consistent solution vector.
- yp0_mod – consistent derivative vector.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_ILL_INPUT – The function was not called before the first call to *IDASolve()*.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

Notes: If the consistent solution vector or consistent derivative vector is not desired, pass NULL for the corresponding argument.

> **Warning:** The user must allocate space for yy0_mod and yp0_mod (if not NULL).

Rootfinding optional output functions

There are two optional output functions associated with rootfinding.

int **IDAGetRootInfo**(void *ida_mem, int *rootsfound)
The function *IDAGetRootInfo()* returns an array showing which functions were found to have a root.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- rootsfound – array of length nrtfn with the indices of the user functions g_i found to have a root. For i $= 0, \ldots,$ nrtfn $- 1$, rootsfound[i] $\neq 0$ if g_i has a root, and $= 0$ if not.

Return value:

- IDA_SUCCESS – The optional output values have been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

Notes: Note that, for the components g_i for which a root was found, the sign of rootsfound[i] indicates the direction of zero-crossing. A value of $+1$ indicates that g_i is increasing, while a value of -1 indicates a decreasing g_i.

Warning: The user must allocate memory for the vector rootsfound.

int **IDAGetNumGEvals**(void *ida_mem, long int *ngevals)
The function *IDAGetNumGEvals()* returns the cumulative number of calls to the user root function g.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- ngevals – number of calls to the user's function g so far.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

IDALS linear solver interface optional output functions

The following optional outputs are available from the IDALS modules:

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

Arguments:

- ida_mem – pointer to the IDAS solver object.
- lenrwLS – the number of real values in the IDALS workspace.
- leniwLS – the number of integer values in the IDALS workspace.

Return value:

- IDALS_SUCCESS – The optional output value has been successfully set.
- IDALS_MEM_NULL – The ida_mem pointer is NULL.

- IDALS_LMEM_NULL – The IDALS 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 IDALS is not included in this report.

Warning: The previous routines IDADlsGetWorkspace() and IDASpilsGetWorkspace() 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.

int **IDAGetNumJacEvals**(void *ida_mem, long int *njevals)

The function *IDAGetNumJacEvals()* returns the cumulative number of calls to the IDALS Jacobian approximation function.

Arguments:

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

Return value:

- IDALS_SUCCESS – The optional output value has been successfully set.
- IDALS_MEM_NULL – The ida_mem pointer is NULL.
- IDALS_LMEM_NULL – The IDALS linear solver has not been initialized.

Warning: The previous routine IDADlsGetNumJacEvals() 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.

int **IDAGetNumLinResEvals**(void *ida_mem, long int *nrevalsLS)

The function *IDAGetNumLinResEvals()* 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:

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

Return value:

- IDALS_SUCCESS – The optional output value has been successfully set.
- IDALS_MEM_NULL – The ida_mem pointer is NULL.
- IDALS_LMEM_NULL – The IDALS 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 IDADlsGetNumRhsEvals() and IDASpilsGetNumRhsEvals() are now deprecated.

int **IDAGetNumLinIters**(void *ida_mem, long int *nliters)

The function *IDAGetNumLinIters()* returns the cumulative number of linear iterations.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `nliters` – the current number of linear iterations.

Return value:

- `IDALS_SUCCESS` – The optional output value has been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.

Warning: The previous routine `IDASpilsGetNumLinIters()` 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.

int **IDAGetNumLinConvFails**(void *ida_mem, long int *nlcfails)

The function *IDAGetNumLinConvFails()* returns the cumulative number of linear convergence failures.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `nlcfails` – the current number of linear convergence failures.

Return value:

- `IDALS_SUCCESS` – The optional output value has been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.

Warning: The previous routine `IDASpilsGetNumConvFails()` 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.

int **IDAGetNumPrecEvals**(void *ida_mem, long int *npevals)

The function *IDAGetNumPrecEvals()* returns the cumulative number of preconditioner evaluations, i.e., the number of calls made to `psetup`.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `npevals` – the cumulative number of calls to `psetup`.

Return value:

- `IDALS_SUCCESS` – The optional output value has been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.

Warning: The previous routine `IDASpilsGetNumPrecEvals()` 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.

int **IDAGetNumPrecSolves**(void *ida_mem, long int *npsolves)

The function *IDAGetNumPrecSolves()* returns the cumulative number of calls made to the preconditioner solve function, `psolve`.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `npsolves` – the cumulative number of calls to `psolve`.

Return value:

- `IDALS_SUCCESS` – The optional output value has been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.

Warning: The previous routine `IDASpilsGetNumPrecSolves()` 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.

int **IDAGetNumJTSetupEvals**(void *ida_mem, long int *njtsetup)

The function *IDAGetNumJTSetupEvals()* returns the cumulative number of calls made to the Jacobian-vector product setup function `jtsetup`.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `njtsetup` – the current number of calls to `jtsetup`.

Return value:

- `IDALS_SUCCESS` – The optional output value has been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.

Warning: The previous routine `IDASpilsGetNumJTSetupEvals()` 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.

int **IDAGetNumJtimesEvals**(void *ida_mem, long int *njvevals)

The function *IDAGetNumJtimesEvals()* returns the cumulative number of calls made to the Jacobian-vector product function, `jtimes`.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `njvevals` – the cumulative number of calls to `jtimes`.

Return value:

- `IDALS_SUCCESS` – The optional output value has been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.

Warning: The previous routine `IDASpilsGetNumJtimesEvals()` 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.

int **IDAGetLastLinFlag**(void *ida_mem, long int *lsflag)

The function *IDAGetLastLinFlag()* returns the last return value from an IDALS routine.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `lsflag` – the value of the last return flag from an IDALS function.

Return value:

- `IDALS_SUCCESS` – The optional output value has been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.

Notes: If the IDALS setup function failed (i.e., *IDASolve()* returned `IDA_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 IDALS 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 IDALS solve function failed (*IDASolve()* returned `IDA_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 `IDADlsGetLastFlag()` and `IDASpilsGetLastFlag()` 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.

char ***IDAGetLinReturnFlagName**(long int lsflag)

The function *IDAGetLinReturnFlagName()* returns the name of the IDALS constant corresponding to `lsflag`.

Arguments:

- `flag` – the flag returned by a call to an IDAS 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 `IDADlsGetReturnFlagName()` and `IDASpilsGetReturnFlagName()` 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.

5.1.4.13 IDAS reinitialization function

The function *IDAReInit()* reinitializes the main IDAS solver for the solution of a new problem, where a prior call to *IDAInit()* has been made. The new problem must have the same size as the previous one. *IDAReInit()* performs the same input checking and initializations that *IDAInit()* does, but does no memory allocation, as it assumes that the existing internal memory is sufficient for the new problem. A call to *IDAReInit()* deletes the solution history that was stored internally during the previous integration. Following a successful call to *IDAReInit()*, call *IDASolve()* again for the solution of the new problem.

The use of *IDAReInit()* requires that the maximum method order, `maxord`, is no larger for the new problem than for the problem specified in the last call to *IDAInit()*. In addition, the same `N_Vector` module set for the previous problem will be reused for the new problem.

If there are changes to the linear solver specifications, make the appropriate calls to either the linear solver objects themselves, or to the IDALS interface routines, as described in §5.1.4.5.

If there are changes to any optional inputs, make the appropriate `IDASet***` calls, as described in §5.1.4.10. Otherwise, all solver inputs set previously remain in effect.

One important use of the *IDAReInit()* function is in the treating of jump discontinuities in the residual function. Except in cases of fairly small jumps, it is usually more efficient to stop at each point of discontinuity and restart the integrator with a readjusted DAE model, using a call to *IDAReInit()*. To stop when the location of the discontinuity is known, simply make that location a value of t_{out}. To stop when the location of the discontinuity is determined by the solution, use the rootfinding feature. In either case, it is critical that the residual function *not* incorporate the discontinuity, but rather have a smooth extention over the discontinuity, so that the step across it (and subsequent rootfinding, if used) can be done efficiently. Then use a switch within the residual function (communicated through `user_data`) that can be flipped between the stopping of the integration and the restart, so that the restarted problem uses the new values (which have jumped). Similar comments apply if there is to be a jump in the dependent variable vector.

int **IDAReInit**(void *ida_mem, *realtype* t0, *N_Vector* y0, *N_Vector* yp0)

The function *IDAReInit()* provides required problem specifications and reinitializes IDAS.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `t0` – is the initial value of t.
- `y0` – is the initial value of y.
- `yp0` – is the initial value of $\dot{y}$.

Return value:

- `IDA_SUCCESS` – The call to was successful.
- `IDA_MEM_NULL` – The IDAS solver object was not initialized through a previous call to *IDACreate()*.
- `IDA_NO_MALLOC` – Memory space for the IDAS solver object was not allocated through a previous call to *IDAInit()*.
- `IDA_ILL_INPUT` – An input argument to *IDAReInit()* has an illegal value.

Notes: If an error occurred, *IDAReInit()* also sends an error message to the error handler function.

5.1.5 User-supplied functions

The user-supplied functions consist of one function defining the DAE residual, (optionally) a function that handles error and warning messages, (optionally) a function that provides the error weight vector, (optionally) one or two functions that provide Jacobian-related information for the linear solver, and (optionally) one or two functions that define the preconditioner for use in any of the Krylov iteration algorithms.

5.1.5.1 DAE residual function

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

typedef int (***IDAResFn**)(*realtype* tt, *N_Vector* yy, *N_Vector* yp, *N_Vector* rr, void *user_data)

This function computes the problem residual for given values of the independent variable t, state vector y, and derivative $\dot{y}$.

Arguments:

- `tt` – is the current value of the independent variable.
- `yy` – is the current value of the dependent variable vector, $y(t)$.
- `yp` – is the current value of $\dot{y}(t)$.
- `rr` – is the output residual vector $F(t, y, \dot{y})$.
- `user_data` – is a pointer to user data, the same as the `user_data` pointer parameter passed to *IDASetUserData()*.

Return value: An *IDAResFn* function type should return a value of 0 if successful, a positive value if a recoverable error occurred (e.g., `yy` has an illegal value), 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: A recoverable failure error return from the *IDAResFn* is typically used to flag a value of the dependent variable y that is "illegal" in some way (e.g., negative where only a non-negative value is physically meaningful). If such a return is made, IDAS will attempt to recover (possibly repeating the nonlinear solve, or reducing the step size) in order to avoid this recoverable error return.

For efficiency reasons, the DAE residual function is not evaluated at the converged solution of the nonlinear solver. Therefore, in general, a recoverable error in that converged value cannot be corrected. (It may be detected when the residual function is called the first time during the following integration step, but a successful step cannot be undone.)

However, if the user program also includes quadrature integration, the state variables can be checked for legality in the call to *IDAQuadRhsFn*, which is called at the converged solution of the nonlinear system, and therefore IDAS can be flagged to attempt to recover from such a situation. Also, if sensitivity analysis is performed with the staggered method, the DAE residual function is called at the converged solution of the nonlinear system, and a recoverable error at that point can be flagged, and IDAS will then try to correct it.

5.1.5.2 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 *IDASetErrFile()*), the user may provide a function of type *IDAErrHandlerFn* to process any such messages. The function type *IDAErrHandlerFn* is defined as follows:

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

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

Arguments:

- `error_code` – is the error code.
- `module` – is the name of the IDAS 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 *IDASetErrHandlerFn()*.

Return value: This function has no return value.

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

5.1.5.3 Error weight function

typedef int (***IDAEwtFn**)(*N_Vector* y, *N_Vector* ewt, void *user_data)

This function computes the WRMS error weights for the vector y.

Arguments:

- `y` – is the value of the dependent variable vector at which the weight vector is to be computed.
- `ewt` – is the output vector containing the error weights.
- `user_data` – is a pointer to user data, the same as the `user_data` parameter passed to *IDASetUserData()*.

Return value:

- `0` – if it the error weights were successfully set.
- `-1` – if any error occured.

Notes: Allocation of memory for `ewt` is handled within IDAS.

Warning: The error weight vector must have all components positive. It is the user's responsiblity to perform this test and return -1 if it is not satisfied.

5.1.5.4 Rootfinding function

If a rootfinding problem is to be solved during the integration of the DAE system, the user must supply a function of type *IDARootFn*, defined as follows:

typedef int (***IDARootFn**)(*realtype* t, *N_Vector* y, *N_Vector* yp, *realtype* *gout, void *user_data)

This function computes a vector-valued function $g(t, y, \dot{y})$ such that the roots of the `nrtfn` components $g_i(t, y, \dot{y})$ are to be found during the integration.

Arguments:

- `t` – is the current value of the independent variable.
- `y` – is the current value of the dependent variable vector, $y(t)$.
- `yp` – is the current value of $\dot{y}(t)$, the $t-$ derivative of y.
- `gout` – is the output array, of length `nrtfn`, with components $g_i(t, y, \dot{y})$.
- `user_data` – is a pointer to user data, the same as the `user_data` parameter passed to *IDASetUserData()*.

Return value: `0` if successful or non-zero if an error occured (in which case the integration is halted and *IDASolve()* returns `IDA_RTFUNC_FAIL`).

Notes: Allocation of memory for `gout` is handled within IDAS.

5.1.5.5 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 *IDASetLinearSolver()*), the user may provide a function of type *IDALsJacFn* defined as follows:

typedef int (***IDALsJacFn**)(*realtype* t, *realtype* c_j, *N_Vector* y, *N_Vector* yp, *N_Vector* r, *SUNMatrix* Jac, void *user_data, *N_Vector* tmp1, *N_Vector* tmp2, *N_Vector* tmp3)

This function computes the Jacobian matrix J of the DAE system (or an approximation to it), defined by (2.6).

Arguments:

- `tt` – is the current value of the independent variable t.
- `cj` – is the scalar in the system Jacobian, proportional to the inverse of the step size (α in (2.6)).
- `yy` – is the current value of the dependent variable vector, $y(t)$.
- `yp` – is the current value of $\dot{y}(t)$.
- `rr` – is the current value of the residual vector $F(t, y, \dot{y})$.
- `Jac` – is the output (approximate) Jacobian matrix (of type `SUNMatrix`), $J = \dfrac{\partial F}{\partial y} + cj\,\dfrac{\partial F}{\partial \dot{y}}$.
- `user_data` - is a pointer to user data, the same as the `user_data` parameter passed to *IDASetUserData()*.
- `tmp1`, `tmp2`, and `tmp3` – are pointers to memory allocated for variables of type `N_Vector` which can be used by *IDALsJacFn()* function as temporary storage or work space.

Return value: An *IDALsJacFn* should return 0 if successful, a positive value if a recoverable error occurred, or a negative value if a nonrecoverable error occurred.

In the case of a recoverable eror return, the integrator will attempt to recover by reducing the stepsize, and hence changing α in (2.6).

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 $J(t, y, \dot{y})$ is zeroed out prior to calling the user-supplied Jacobian function so only nonzero elements need to be loaded into Jac.

With the default nonlinear solver (the native SUNDIALS Newton method), each call to the user's *IDALsJacFn()* function is preceded by a call to the *IDAResFn()* user function with the same $(t, y, \dot{y})$ arguments. Thus the Jacobian function can use any auxiliary data that is computed and saved during the evaluation of the DAE residual. In the case of a user-supplied or external nonlinear solver, this is also true if the residual function is evaluated prior to calling the linear solver setup function (see §9.1.4 for more information).

If the user's *IDALsJacFn* function uses difference quotient approximations, it may need to access quantities not in the call list. These quantities may include the current stepsize, the error weights, etc. To obtain these, the user will need to add a pointer to ida_mem to user_data and then use the IDAGet* functions described in §5.1.4.12. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundials_types.h.

dense:

A user-supplied dense Jacobian function must load the Neq $\times$ Neq dense matrix Jac with an approximation to the Jacobian matrix $J(t, y, \dot{y})$ at the point (tt, yy, yp). The accessor macros SM_ELEMENT_D and SM_COLUMN_D allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the SUNMATRIX_DENSE type. SM_ELEMENT_D(J, i, j) references the (i, j)-th element of the dense matrix Jac (with i, j$= 0 \ldots$ N-1). This macro is meant for small problems for which efficiency of access is not a major concern. Thus, in terms of the indices m and n ranging from 1 to N, the Jacobian element $J_{m,n}$ can be set using the statement SM_ELEMENT_D(J, m-1, n-1) = $J_{m,n}$. Alternatively, SM_COLUMN_D(J, j) returns a pointer to the first element of the j-th column of Jac (with j$= 0 \ldots$ N $- 1$), and the elements of the j-th column can then be accessed using ordinary array indexing. Consequently, $J_{m,n}$ can be loaded using the statements col_n = SM_COLUMN_D(J, n-1); col_n[m-1] = $J_{m,n}$. For large problems, it is more efficient to use SM_COLUMN_D than to use SM_ELEMENT_D. Note that both of these macros number rows and columns starting from 0. The SUNMATRIX_DENSE type and accessor macros are documented in §7.3.

banded:

A user-supplied banded Jacobian function must load the Neq $\times$ Neq banded matrix Jac with an approximation to the Jacobian matrix $J(t, y, \dot{y})$ at the point (tt, yy, yp). The accessor macros SM_ELEMENT_B, SM_COLUMN_B, and SM_COLUMN_ELEMENT_B allow the user to read and write banded matrix elements without making specific references to the underlying representation of the SUNMATRIX_BAND type. SM_ELEMENT_B(J, i, j) references the (i, j)-th element of the banded matrix Jac, counting from 0. This macro is meant for use in small problems for which efficiency of access is not a major concern. Thus, in terms of the indices m and n ranging from 1 to N with (m, n) within the band defined by mupper and mlower, the Jacobian element $J_{m,n}$ can be loaded using the statement SM_ELEMENT_B(J, m-1, n-1) = $J_{m,n}$. The elements within the band are those with -mupper $\leq$ m-n $\leq$ mlower. Alternatively, SM_COLUMN_B(J, j) returns a pointer to the diagonal element of the j-th column of Jac, and if we assign this address to realtype *col_j, then the i-th element of the j-th column is given by SM_COLUMN_ELEMENT_B(col_j, i, j), counting from 0. Thus, for (m, n) within the band, $J_{m,n}$ can be loaded by setting col_n = SM_COLUMN_B(J, n-1); and SM_COLUMN_ELEMENT_B(col_n, m-1, n-1) = $J_{m,n}$. The elements of the j-th column can also be accessed via ordinary array indexing, but this approach requires knowledge of the underlying storage for a band matrix of type SUNMATRIX_BAND. The array col_n can be indexed from $-$mupper to mlower. For large problems, it is more efficient to use SM_COLUMN_B and SM_COLUMN_ELEMENT_B than to use the SM_ELEMENT_B macro. As in the dense case, these macros all number rows and columns starting from 0. The SUNMATRIX_BAND type and accessor macros are documented in §7.6.

sparse:

A user-supplied sparse Jacobian function must load the `Neq` $\times$ `Neq` compressed-sparse-column or compressed-sparse-row matrix `Jac` with an approximation to the Jacobian matrix $J(t, y, \dot{y})$ at the point (`tt`, `yy`, `yp`). Storage for `Jac` already exists on entry to this function, although the user should ensure that sufficient space is allocated in `Jac` to hold the nonzero values to be set; if the existing space is insufficient the user may reallocate the data and index arrays as needed. The amount of allocated space in a `SUNMATRIX_SPARSE` object may be accessed using the macro `SM_NNZ_S` or the routine `SUNSparseMatrix_NNZ`. The `SUNMATRIX_SPARSE` type and accessor macros are documented in §7.8.

Warning: The previous function type `IDADlsJacFn()` is identical to *IDALsJacFn()*, 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.

5.1.5.6 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 *IDASetLinearSolver()*), the user may provide a function of type *IDALsJacTimesVecFn* 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 (***IDALsJacTimesVecFn**)(*realtype* tt, *N_Vector* yy, *N_Vector* yp, *N_Vector* rr, *N_Vector* v, *N_Vector* Jv, *realtype* cj, void *user_data, *N_Vector* tmp1, *N_Vector* tmp2)

This function computes the product Jv of the DAE system Jacobian J (or an approximation to it) and a given vector `v`, where J is defined by (2.6).

Arguments:

- `tt` – is the current value of the independent variable.
- `yy` – is the current value of the dependent variable vector, $y(t)$.
- `yp` – is the current value of $\dot{y}(t)$.
- `rr` – is the current value of the residual vector $F(t, y, \dot{y})$.
- `v` – is the vector by which the Jacobian must be multiplied to the right.
- `Jv` – is the computed output vector.
- `cj` – is the scalar in the system Jacobian, proportional to the inverse of the step size (α in (2.6)).
- `user_data` – is a pointer to user data, the same as the `user_data` parameter passed to *IDASetUserData()*.
- `tmp1` and `tmp2` – are pointers to memory allocated for variables of type `N_Vector` which can be used by *IDALsJacTimesVecFn* as temporary storage or work space.

Return value: The value returned by the Jacobian-times-vector function should be 0 if successful. A nonzero value indicates that a nonrecoverable error occurred.

Notes: This function must return a value of Jv that uses an approximation to the **current** value of J, i.e. as evaluated at the current $(t, y, \dot{y})$.

If the user's *IDALsJacTimesVecFn()* function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, the user will need to add a pointer to `ida_mem` to `user_data` and then use the `IDAGet*` functions described in §5.1.4.12. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

Warning: The previous function type `IDASpilsJacTimesVecFn()` is identical to *IDALsJacTimesVecFn()*, 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.

5.1.5.7 Jacobian-vector product setup (matrix-free linear solvers)

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

typedef int (***IDALsJacTimesSetupFn**)(*realtype* tt, *N_Vector* yy, *N_Vector* yp, *N_Vector* rr, ealtype cj, void *user_data);

This function setups any data needed by Jv product function (see *IDALsJacTimesVecFn*).

Arguments:

- `tt` – is the current value of the independent variable.
- `yy` – is the current value of the dependent variable vector, $y(t)$.
- `yp` – is the current value of $\dot{y}(t)$.
- `rr` – is the current value of the residual vector $F(t, y, \dot{y})$.
- `cj` – is the scalar in the system Jacobian, proportional to the inverse of the step size (α in (2.6)).
- `user_data` – is a pointer to user data, the same as the `user_data` parameter passed to *IDASetUserData()*.

Return value: The value returned by the Jacobian-vector 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: Each call to the Jacobian-vector product setup function is preceded by a call to the *IDAResFn* user function with the same $(t, y, \dot{y})$ arguments. Thus, the setup function can use any auxiliary data that is computed and saved during the evaluation of the DAE residual.

If the user's *IDALsJacTimesVecFn* function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, the user will need to add a pointer to `ida_mem` to `user_data` and then use the `IDAGet*` functions described in §5.1.4.12. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

Warning: The previous function type `IDASpilsJacTimesSetupFn()` is identical to *IDALsJacTimesSetupFn()*, 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.

5.1.5.8 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 a left preconditioner matrix which approximates (at least crudely) the Jacobian matrix $J = \partial F/\partial y + cj\ \partial F/\partial \dot{y}$. This function must be of type *IDALsPrecSolveFn*, defined as follows:

typedef int (***IDALsPrecSolveFn**)(*realtype* tt, *N_Vector* yy, *N_Vector* yp, *N_Vector* rr, *N_Vector* rvec, *N_Vector* zvec, *realtype* cj, *realtype* delta, void *user_data)

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

Arguments:

- `tt` – is the current value of the independent variable.
- `yy` – is the current value of the dependent variable vector, $y(t)$.
- `yp` – is the current value of $\dot{y}(t)$.
- `rr` – is the current value of the residual vector $F(t, y, \dot{y})$.
- `rvec` – is the right-hand side vector r of the linear system to be solved.
- `zvec` – is the computed output vector.
- `cj` – is the scalar in the system Jacobian, proportional to the inverse of the step size (α in (2.6)).
- `delta` – is an input tolerance to be used if an iterative method is employed in the solution. In that case, the residual vector $Res = r - Pz$ of the system should be made less than `delta` in weighted l_2 norm, i.e., $\sqrt{\sum_i (Res_i \cdot ewt_i)^2} <$ `delta`. To obtain the `N_Vector ewt`, call *IDAGetErrWeights()*.
- `user_data` – is a pointer to user data, the same as the `user_data` parameter passed to *IDASetUserData()*.

Return value: The value returned by the preconditioner solve 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).

5.1.5.9 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 *IDALsPrecSetupFn*, defined as follows:

typedef int (***IDALsPrecSetupFn**)(*realtype* tt, *N_Vector* yy, *N_Vector* yp, *N_Vector* rr, *realtype* cj, void *user_data)

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

Arguments:

- `tt` – is the current value of the independent variable.
- `yy` – is the current value of the dependent variable vector, $y(t)$.
- `yp` – is the current value of $\dot{y}(t)$.
- `rr` – is the current value of the residual vector $F(t, y, \dot{y})$.
- `cj` – is the scalar in the system Jacobian, proportional to the inverse of the step size (α in (2.6)).
- `user_data` – is a pointer to user data, the same as the `user_data` parameter passed to *IDASetUserData()*.

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: With the default nonlinear solver (the native SUNDIALS Newton method), each call to the preconditioner setup function is preceded by a call to the *IDAResFn* user function with the same $(t, y, \dot{y})$ arguments. Thus the preconditioner setup function can use any auxiliary data that is computed and saved during the evaluation of the DAE residual. In the case of a user-supplied or external nonlinear solver, this is also true if the residual function is evaluated prior to calling the linear solver setup function (see §9.1.4 for more information).

This function is not called in advance of every call to the preconditioner solve function, but rather is called only as often as needed to achieve convergence in the nonlinear solver.

If the user's *IDALsPrecSetupFn* function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, the user will need to add a pointer to `ida_mem` to `user_data` and then use the `IDAGet*` functions described in §5.1.4.12. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

5.2 Integration of pure quadrature equations

IDA allows the DAE system to include *pure quadratures*. In this case, it is more efficient to treat the quadratures separately by excluding them from the nonlinear solution stage. To do this, begin by excluding the quadrature variables from the vectors `yy` and `yp` and the quadrature equations from within `res`. Thus a separate vector `yQ` of quadrature variables is to satisfy $(\mathrm{d}/\mathrm{d}t)\texttt{yQ} = f_Q(t, y, \dot{y})$. The following is an overview of the sequence of calls in a user's main program in this situation. Steps that have changed from the skeleton program presented in §5.1.3 are bolded.

1. Initialize parallel or multi-threaded environment, if appropriate
2. Create the SUNDIALS context object with *SUNContext_Create()*
3. Set vector of initial values
4. Create matrix object
5. Create linear solver object
6. Create nonlinear solver object
7. Create IDAS object
8. Initialize IDAS solver
9. Specify integration tolerances
10. Set linear solver optional inputs
11. Attach linear solver module
12. Attach nonlinear solver module
13. Set nonlinear solver optional inputs
14. **Set vector of initial values for quadrature variables**

    Typically, the quadrature variables should be initialized to 0.
15. **Initialize quadrature integration**

    Call *IDAQuadInit()* to specify the quadrature equation right-hand side function and to allocate internal memory related to quadrature integration. See §5.2.1 for details.
16. **Set optional inputs for quadrature integration**

    Call *IDASetQuadErrCon()* to indicate whether or not quadrature variables shoule be used in the step size control mechanism, and to specify the integration tolerances for quadrature variables. See §5.2.4 for details.
17. Specify rootfinding problem
18. Set optional inputs
19. Correct initial values
20. Advance solution in time
21. **Extract quadrature variables**

    Call *IDAGetQuad()* to obtain the values of the quadrature variables at the current time.

22. Get optional outputs
23. **Get quadrature optional outputs**

    Call `IDAGetQuad**` functions to obtain optional output related to the integration of quadratures. See §5.2.5 for details.
24. Deallocate memory
25. Finalize MPI, if used

5.2.1 Quadrature initialization and deallocation functions

The function *IDAQuadInit()* activates integration of quadrature equations and allocates internal memory related to these calculations. The form of the call to this function is as follows:

int **IDAQuadInit**(void *ida_mem, *IDAQuadRhsFn* rhsQ, *N_Vector* yQ0)

The function *IDAQuadInit()* provides required problem specifications, allocates internal memory, and initializes quadrature integration.

Arguments:

- `ida_mem` – pointer to the IDA memory block returned by *IDACreate()*.
- `rhsQ` – is the C function which computes f_Q , the right-hand side of the quadrature equations. This function has the form `f(Qt, yy, yp, rhsQ, user_data)` for full details see §5.2.6.
- `yQ0` – is the initial value of y_Q.

Return value:

- `IDA_SUCCESS` – The call to *IDAQuadInit()* was successful.
- `IDA_MEM_NULL` – The IDA memory was not initialized by a prior call to *IDACreate()*.
- `IDA_MEM_FAIL` – A memory allocation request failed.

Notes:

If an error occurred, *IDAQuadInit()* also sends an error message to the error handler function.

In terms of the number of quadrature variables, N_q, and maximum method order, `maxord`, the size of the real and integer workspaces are increased by $(\texttt{maxord} + 5)N_q$. If *IDAQuadSVtolerances()* is called, the workspaces are further increased by N_q.

The function *IDAQuadReInit()*, useful during the solution of a sequence of problems of same size, reinitializes the quadrature-related internal memory and must follow a call to *IDAQuadInit()* (and maybe a call to *IDAReInit()*). The number N_q of quadratures is assumed to be unchanged from the prior call to *IDAQuadInit()*. The call to the *IDAQuadReInit()* function has the following form:

int **IDAQuadReInit**(void *ida_mem, *N_Vector* yQ0)

The function *IDAQuadReInit()* provides required problem specifications and reinitializes the quadrature integration.

Arguments:

- `ida_mem` – pointer to the IDA memory block.
- `yQ0` – is the initial value of y_Q.

Return value:

- `IDA_SUCCESS` – The call to *IDAReInit()* was successful.
- `IDA_MEM_NULL` – The IDA memory was not initialized by a prior call to *IDACreate()*.

- IDA_NO_QUAD – Memory space for the quadrature integration was not allocated by a prior call to *IDAQuadInit()*.

Notes: If an error occurred, *IDAQuadReInit()* also sends an error message to the error handler function.

void **IDAQuadFree**(void *ida_mem)

The function *IDAQuadFree()* frees the memory allocated for quadrature integration.

Arguments:

- ida_mem – pointer to the IDA memory block.

Return value:

- The function has no return value.

Notes: In general, *IDAQuadFree()* need not be called by the user as it is invoked automatically by *IDAFree()*.

5.2.2 IDAS solver function

Even if quadrature integration was enabled, the call to the main solver function *IDASolve()* is exactly the same. However, in this case the return value flag can also be one of the following:

- IDA_QRHS_FAIL – The quadrature right-hand side function failed in an unrecoverable manner.
- IDA_FIRST_QRHS_ERR – The quadrature right-hand side function failed at the first call.
- IDA_REP_QRHS_ERR – Convergence test failures occurred too many times due to repeated recoverable errors in the quadrature right-hand side function. This value will also be returned if the quadrature right-hand side function had repeated recoverable errors during the estimation of an initial step size (assuming the quadrature variables are included in the error tests).

5.2.3 Quadrature extraction functions

If quadrature integration has been initialized by a call to *IDAQuadInit()*, or reinitialized by a call to *IDAQuadReInit()*, then IDA computes both a solution and quadratures at time t. However, *IDASolve()* will still return only the solution y in y. Solution quadratures can be obtained using the following function:

int **IDAGetQuad**(void *ida_mem, *realtype* tret, *N_Vector* yQ)

The function *IDAGetQuad()* returns the quadrature solution vector after a successful return from *IDASolve()*.

Arguments:

- ida_mem – pointer to the memory previously allocated by *IDAInit()*.
- tret – the time reached by the solver output.
- yQ – the computed quadrature vector.

Return value:

- IDA_SUCCESS – *IDAGetQuad()* was successful.
- IDA_MEM_NULL – ida_mem was NULL.
- IDA_NO_QUAD – Quadrature integration was not initialized.
- IDA_BAD_DKY – yQ is NULL.

The function *IDAGetQuadDky()* computes the k-th derivatives of the interpolating polynomials for the quadrature variables at time t. This function is called by *IDAGetQuad()* with k = 0 and with the current time at which *IDASolve()* has returned, but may also be called directly by the user.

int **IDAGetQuadDky**(void *ida_mem, *realtype* t, int k, *N_Vector* dkyQ)

The function *IDAGetQuadDky()* returns derivatives of the quadrature solution vector after a successful return from IDA.

Arguments:

- ida_mem – pointer to the memory previously allocated by *IDAInit()*.
- t – the time at which quadrature information is requested. The time t must fall within the interval defined by the last successful step taken by IDAS.
- k – order of the requested derivative. This must be $\leq$ klast.
- dkyQ – the vector containing the derivative. This vector must be allocated by the user.

Return value:

- IDA_SUCCESS – *IDAGetQuadDky()* succeeded.
- IDA_MEM_NULL – The pointer to ida_mem was NULL.
- IDA_NO_QUAD – Quadrature integration was not initialized.
- IDA_BAD_DKY – The vector dkyQ is NULL.
- IDA_BAD_K – k is not in the range $0, 1, \ldots,$ klast.
- IDA_BAD_T – The time t is not in the allowed range.

Notes: In case of an error return, an error message is also sent to the error handler function.

5.2.4 Optional inputs for quadrature integration

IDA provides the following optional input functions to control the integration of quadrature equations.

int **IDASetQuadErrCon**(void *ida_mem, *booleantype* errconQ)

The function *IDASetQuadErrCon()* specifies whether or not the quadrature variables are to be used in the step size control mechanism within IDA. If they are, the user must call either *IDAQuadSStolerances()* or *IDAQuadSVtolerances()* to specify the integration tolerances for the quadrature variables.

Arguments:

- ida_mem – pointer to the IDA memory block.
- errconQ – specifies whether quadrature variables are included SUNTRUE or not SUNFALSE in the error control mechanism.

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_NO_QUAD – Quadrature integration has not been initialized.

Notes: By default, errconQ is set to SUNFALSE.

> **Warning:** It is illegal to call *IDASetQuadErrCon()* before a call to *IDAQuadInit()*.

If the quadrature variables are part of the step size control mechanism, one of the following functions must be called to specify the integration tolerances for quadrature variables.

int **IDAQuadSStolerances**(void *ida_mem, *realtype* reltolQ, *realtype* abstolQ)
The function *IDAQuadSStolerances()* specifies scalar relative and absolute tolerances.

Arguments:

- `ida_mem` – pointer to the IDA memory block.
- `reltolQ` – tolerances is the scalar relative error tolerance.
- `abstolQ` – is the scalar absolute error tolerance.

Return value:

- `IDA_SUCCESS` – The optional value has been successfully set.
- `IDA_NO_QUAD` – Quadrature integration was not initialized.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDA_ILL_INPUT` – One of the input tolerances was negative.

int **IDAQuadSVtolerances**(void *ida_mem, *realtype* reltolQ, *N_Vector* abstolQ)
The function *IDAQuadSVtolerances()* specifies scalar relative and vector absolute tolerances.

Arguments:

- `ida_mem` – pointer to the IDA memory block.
- `reltolQ` – tolerances is the scalar relative error tolerance.
- `abstolQ` – is the vector absolute error tolerance.

Return value:

- `IDA_SUCCESS` – The optional value has been successfully set.
- `IDA_NO_QUAD` – Quadrature integration was not initialized.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDA_ILL_INPUT` – One of the input tolerances was negative.

5.2.5 Optional outputs for quadrature integration

IDA provides the following functions that can be used to obtain solver performance information related to quadrature integration.

int **IDAGetQuadNumRhsEvals**(void *ida_mem, long int *nrhsQevals)
The function *IDAGetQuadNumRhsEvals()* returns the number of calls made to the user's quadrature right-hand side function.

Arguments:

- `ida_mem` – pointer to the IDA memory block.
- `nrhsQevals` – number of calls made to the user's `rhsQ` function.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDA_NO_QUAD` – Quadrature integration has not been initialized.

int **IDAGetQuadNumErrTestFails**(void *ida_mem, long int *nQetfails)

The function *IDAGetQuadNumErrTestFails()* returns the number of local error test failures due to quadrature variables.

Arguments:

- `ida_mem` – pointer to the IDA memory block.
- `nQetfails` – number of error test failures due to quadrature variables.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDA_NO_QUAD` – Quadrature integration has not been initialized.

int **IDAGetQuadErrWeights**(void *ida_mem, *N_Vector* eQweight)

The function *IDAGetQuadErrWeights()* returns the quadrature error weights at the current time.

Arguments:

- `ida_mem` – pointer to the IDA memory block.
- `eQweight` – quadrature error weights at the current time.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDA_NO_QUAD` – Quadrature integration has not been initialized.

> **Warning:** The user must allocate memory for `eQweight`. If quadratures were not included in the error control mechanism (through a call to *IDASetQuadErrCon()* with `errconQ = SUNTRUE`), *IDAGetQuadErrWeights()* does not set the `eQweight` vector.

int **IDAGetQuadStats**(void *ida_mem, long int *nrhsQevals, long int *nQetfails)

The function *IDAGetQuadStats()* returns the IDAS integrator statistics as a group.

Arguments:

- `ida_mem` – pointer to the IDA memory block.
- `nrhsQevals` – number of calls to the user's `rhsQ` function.
- `nQetfails` – number of error test failures due to quadrature variables.

Return value:

- `IDA_SUCCESS` – the optional output values have been successfully set.
- `IDA_MEM_NULL` – the `ida_mem` pointer is `NULL`.
- `IDA_NO_QUAD` – Quadrature integration has not been initialized.

5.2.6 User-supplied function for quadrature integration

For integration of quadrature equations, the user must provide a function that defines the right-hand side of the quadrature equations (in other words, the integrand function of the integral that must be evaluated). This function must be of type *IDAQuadRhsFn()* defined as follows:

typedef int (***IDAQuadRhsFn**)(*realtype* tres, *N_Vector* yy, *N_Vector* yp, *N_Vector* rrQ, void *user_data)

This function computes the quadrature equation right-hand side for a given value of the independent variable t and state vectors y and $\dot{y}$.

Arguments:

- `t` – is the current value of the independent variable.
- `yy` – is the current value of the dependent variable vector, $y(t)$.
- `yp` – is the current value of the dependent variable derivative vector, $\dot{y}(t)$.
- `rrQ` – is the output vector $f_Q(t, y, \dot{y})$.
- `user_data` – is the `user_data` pointer passed to *IDASetUserData()* .

Return value:

A *IDAQuadRhsFn()* should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `IDA_QRHS_FAIL` is returned).

Notes:

Allocation of memory for `rhsQ` is automatically handled within IDAS.

Both `y` and `rhsQ` are of type `N_Vector`, but they typically have different internal representations. It is the user's responsibility to access the vector data consistently.

There is one situation in which recovery is not possible even if *IDAQuadRhsFn()* function returns a recoverable error flag. This is when this occurs at the very first call to the *IDAQuadRhsFn()* (in which case IDAS returns `IDA_FIRST_QRHS_ERR`).

5.3 Preconditioner modules

A principal reason for using a parallel DAE solver such as IDAS lies in the solution of partial differential equations (PDEs). Moreover, the use of a Krylov iterative method for the solution of many such problems is motivated by the nature of the underlying linear system of equations (2.5) that must be solved at each time step. The linear algebraic system is large, sparse, and structured. However, if a Krylov iterative method is to be effective in this setting, then a nontrivial preconditioner needs to be used. Otherwise, the rate of convergence of the Krylov iterative method is usually unacceptably slow. Unfortunately, an effective preconditioner tends to be problem-specific.

However, we have developed one type of preconditioner that treats a rather broad class of PDE-based problems. It has been successfully used for several realistic, large-scale problems [34] and is included in a software module within the IDAS package. This module works with the parallel vector module *NVECTOR_PARALLEL* and generates a preconditioner that is a block-diagonal matrix with each block being a band matrix. The blocks need not have the same number of super- and sub-diagonals, and these numbers may vary from block to block. This Band-Block-Diagonal Preconditioner module is called IDABBDPRE.

One way to envision these preconditioners is to think of the domain of the computational PDE problem as being subdivided into M non-overlapping sub-domains. Each of these sub-domains is then assigned to one of the M processors to be used to solve the DAE system. The basic idea is to isolate the preconditioning so that it is local to each processor, and also to use a (possibly cheaper) approximate residual function. This requires the definition of a new function $G(t, y, \dot{y})$ which approximates the function $F(t, y, \dot{y})$ in the definition of the DAE system (2.1). However, the user may

set $G = F$. Corresponding to the domain decomposition, there is a decomposition of the solution vectors y and $\dot{y}$ into M disjoint blocks y_m and $\dot{y}_m$, and a decomposition of G into blocks G_m. The block G_m depends on y_m and $\dot{y}_m$, and also on components of $y_{m'}$ and $\dot{y}_{m'}$ associated with neighboring sub-domains (so-called ghost-cell data). Let $\bar{y}_m$ and $\bar{\dot{y}}_m$ denote y_m and $\dot{y}_m$ (respectively) augmented with those other components on which G_m depends. Then we have

$$G(t, y, \dot{y}) = [G_1(t, \bar{y}_1, \bar{\dot{y}}_1), G_2(t, \bar{y}_2, \bar{\dot{y}}_2), \ldots, G_M(t, \bar{y}_M, \bar{\dot{y}}_M)]^T ,$$

and each of the blocks $G_m(t, \bar{y}_m, \bar{\dot{y}}_m)$ is uncoupled from the others.

The preconditioner associated with this decomposition has the form

$$P = \begin{bmatrix} P_1 & & & \\ & P_2 & & \\ & & \ddots & \\ & & & P_M \end{bmatrix}$$

where

$$P_m \approx \frac{\partial G_m}{\partial y_m} + \alpha \frac{\partial G_m}{\partial \dot{y}_m}$$

This matrix is taken to be banded, with upper and lower half-bandwidths `mudq` and `mldq` defined as the number of non-zero diagonals above and below the main diagonal, respectively. The difference quotient approximation is computed using `mudq` + `mldq` +2 evaluations of G_m, but only a matrix of bandwidth `mukeep` + `mlkeep` +1 is retained.

Neither pair of parameters need be the true half-bandwidths of the Jacobians 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 DAE 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.

The solution of the complete linear system

$$Px = b$$

reduces to solving each of the equations

$$P_m x_m = b_m$$

and this is done by banded LU factorization of P_m followed by a banded backsolve.

Similar block-diagonal preconditioners could be considered with different treatment of the blocks P_m. For example, incomplete LU factorization or an iterative method could be used instead of banded LU factorization.

5.3.1 A parallel band-block-diagonal preconditioner module

The IDABBDPRE module calls two user-provided functions to construct P: a required function `Gres` (of type *IDABBDLocalFn*) which approximates the residual function $G(t, y, \dot{y}) \approx F(t, y, \dot{y})$ and which is computed locally, and an optional function `Gcomm` (of type *IDABBDCommFn*) which performs all inter-process communication necessary to evaluate the approximate residual G. These are in addition to the user-supplied residual function `res`. Both functions take as input the same pointer `user_data` as passed by the user to *IDASetUserData()* and passed to the user's function `res`. The user is responsible for providing space (presumably within `user_data`) for components of `yy` and `yp` that are communicated by `Gcomm` from the other processors, and that are then used by `Gres`, which should not do any communication.

typedef int (***IDABBDLocalFn**)(*sunindextype* Nlocal, *realtype* tt, *N_Vector* yy, *N_Vector* yp, *N_Vector* gval, void *user_data)

This `Gres` function computes $G(t, y, \dot{y})$. It loads the vector `gval` as a function of `tt`, `yy`, and `yp`.

Arguments:

- `Nlocal` – is the local vector length.
- `tt` – is the value of the independent variable.
- `yy` – is the dependent variable.
- `yp` – is the derivative of the dependent variable.
- `gval` – is the output vector.
- `user_data` – is a pointer to user data, the same as the `user_data` parameter passed to *IDASetUserData()*.

Return value:

An *IDABBDLocalFn* function type should return 0 to indicate success, 1 for a recoverable error, or -1 for a non-recoverable error.

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 (***IDABBDCommFn**)(*sunindextype* Nlocal, *realtype* tt, *N_Vector* yy, *N_Vector* yp, void *user_data)

This `Gcomm` function performs all inter-processor communications necessary for the execution of the `Gres` function above, using the input vectors `yy` and `yp`.

Arguments:

- `Nlocal` – is the local vector length.
- `tt` – is the value of the independent variable.
- `yy` – is the dependent variable.
- `yp` – is the derivative of the dependent variable.
- `gval` – is the output vector.
- `user_data` – is a pointer to user data, the same as the `user_data` parameter passed to *IDASetUserData()*.

Return value: An *IDABBDCommFn* function type should return 0 to indicate success, 1 for a recoverable error, or -1 for a non-recoverable error.

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 `res` with the same $(t, y, \dot{y})$ arguments. Thus `Gcomm` can omit any communications done by `res` if relevant to the evaluation of `Gres`. If all necessary communication was done in `res`, then `Gcomm = NULL` can be passed in the call to *IDABBDPrecInit()*.

Besides the header files required for the integration of the DAE problem (see §5.1.2), to use the IDABBDPRE module, the main program must include the header file `ida_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 changed from the user main program presented in §5.1.3 are bolded.

1. Initialize parallel or multi-threaded environment
2. Create the vector of initial values
3. Create matrix object
4. **Create linear solver object**

   When creating the iterative linear solver object, specify the use of left preconditioning (`SUN_PREC_LEFT`) as IDAS only supports left preconditioning.
5. Create nonlinear solver object
6. Create IDAS object
7. Initialize IDAS solver
8. Specify integration tolerances
9. Attach the linear solver
10. **Set linear solver optional inputs**

    Warning: The user should not overwrite the preconditioner setup function or solve function through calls to *IDASetPreconditioner()* optional input function.

11. **Initialize the IDABBDPRE preconditioner module**

    Call *IDABBDPrecInit()* to allocate memory and initialize the internal preconditioner data. The last two arguments of *IDABBDPrecInit()* are the two user-supplied functions described above.
12. Attach nonlinear solver module
13. Set nonlinear solver optional inputs
14. Specify rootfinding problem
15. Set optional inputs
16. Advance solution in time
17. **Get optional outputs**

    Additional optional outputs associated with IDABBDPRE are available by way of two routines described below, *IDABBDPrecGetWorkSpace()* and *IDABBDPrecGetNumGfnEvals()*.
18. Deallocate memory
19. Finalize MPI, if used

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

int **IDABBDPrecInit**(void *ida_mem, *sunindextype* Nlocal, *sunindextype* mudq, *sunindextype* mldq, *sunindextype* mukeep, *sunindextype* mlkeep, *realtype* dq_rel_yy, *IDABBDLocalFn* Gres, *IDABBDCommFn* Gcomm);

The function *IDABBDPrecInit()* initializes and allocates (internal) memory for the IDABBDPRE preconditioner.

Arguments:

- `ida_mem` – pointer to the IDAS solver object.
- `Nlocal` – local vector dimension.
- `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_yy – the relative increment in components of y used in the difference quotient approximations. The default is dq_rel_yy $= \sqrt{\text{unit roundoff}}$, which can be specified by passing dq_rel_yy $= 0.0$.
- Gres – the function which computes the local residual approximation $G(t, y, \dot{y})$.
- Gcomm – the optional function which performs all inter-process communication required for the computation of $G(t, y, \dot{y})$.

Return value:

- IDALS_SUCCESS – The call was successful.
- IDALS_MEM_NULL – The ida_mem pointer was NULL.
- IDALS_MEM_FAIL – A memory allocation request has failed.
- IDALS_LMEM_NULL – An IDALS linear solver memory was not attached.
- IDALS_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 by 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 a 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 on every processor.

The IDABBDPRE module also provides a reinitialization function to allow for a sequence of problems of the same size, with the same linear solver choice, provided there is no change in local_N, mukeep, or mlkeep. After solving one problem, and after calling *IDAReInit()* to re-initialize IDAS for a subsequent problem, a call to *IDABBDPrecReInit()* can be made to change any of the following: the half-bandwidths mudq and mldq used in the difference-quotient Jacobian approximations, the relative increment dq_rel_yy, or one of the user-supplied functions Gres and Gcomm. If there is a change in any of the linear solver inputs, an additional call to the "Set"routines provided by the SUNLinearSolver object, and/or one or more of the corresponding IDASet*** functions, must also be made (in the proper order).

int **IDABBDPrecReInit**(void *ida_mem, *sunindextype* mudq, *sunindextype* mldq, *realtype* dq_rel_yy)

The function *IDABBDPrecReInit()* reinitializes the IDABBDPRE preconditioner.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- 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.
- dq_rel_yy – the relative increment in components of y used in the difference quotient approximations. The default is dq_rel_yy $= \sqrt{\text{unit roundoff}}$, which can be specified by passing dq_rel_yy $= 0.0$.

Return value:

- IDALS_SUCCESS – The call was successful.
- IDALS_MEM_NULL – The ida_mem pointer was NULL.

- IDALS_LMEM_NULL – An IDALS linear solver memory was not attached.
- IDALS_PMEM_NULL – The function *IDABBDPrecInit()* was not previously called.

Notes: If one of the half-bandwidths mudq or mldq is negative or exceeds the value Nlocal - 1, it is replaced by 0 or Nlocal - 1, accordingly.

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

int **IDABBDPrecGetWorkSpace**(void *ida_mem, long int *lenrwBBDP, long int *leniwBBDP)

The function *IDABBDPrecGetWorkSpace()* returns the local sizes of the IDABBDPRE real and integer workspaces.

Arguments:

- ida_mem – pointer to the IDAS solver object.
- lenrwBBDP – local number of real values in the IDABBDPRE workspace.
- leniwBBDP – local number of integer values in the IDABBDPRE workspace.

Return value:

- IDALS_SUCCESS – The optional output value has been successfully set.
- IDALS_MEM_NULL – The ida_mem pointer was NULL.
- IDALS_PMEM_NULL – The IDABBDPRE preconditioner has not been initialized.

Notes: The workspace requirements reported by this routine correspond only to memory allocated within the IDABBDPRE 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 function *IDAGetLinWorkSpace()*.

int **IDABBDPrecGetNumGfnEvals**(void *ida_mem, long int *ngevalsBBDP)

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

Arguments:

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

Return value:

- IDALS_SUCCESS – The optional output value has been successfully set.
- IDALS_MEM_NULL – The ida_mem pointer was NULL.
- IDALS_PMEM_NULL – The IDABBDPRE preconditioner has not been initialized.

In addition to the ngevalsBBDP evaluations of Gres, the costs associated with IDABBDPRE also includes nlinsetups LU factorizations, nlinsetups calls to Gcomm, npsolves banded backsolve calls, and nrevalsLS residual function evaluations, where nlinsetups is an optional IDAS output (see §5.1.4.12), and npsolves and nrevalsLS are linear solver optional outputs (see §5.1.4.12).

5.4 Using IDAS for Forward Sensitivity Analysis

This chapter describes the use of IDAS to compute solution sensitivities using forward sensitivity analysis. One of our main guiding principles was to design the IDAS user interface for forward sensitivity analysis as an extension of that for IVP integration. Assuming a user main program and user-defined support routines for IVP integration have already been defined, in order to perform forward sensitivity analysis the user only has to insert a few more calls into the main program and (optionally) define an additional routine which computes the residual of the sensitivity systems (2.11). The only departure from this philosophy is due to the *IDAResFn* type definition. Without changing the definition of this type, the only way to pass values of the problem parameters to the ODE residual function is to require the user data structure `f_data` to contain a pointer to the array of real parameters p.

IDAS uses various constants for both input and output. These are defined as needed in this chapter, but for convenience are also listed separately in §12.

We begin with a brief overview, in the form of a skeleton user program. Following that are detailed descriptions of the interface to the various user-callable routines and of the user-supplied routines that were not already described in §5.1 or §5.2.

5.4.1 A skeleton of the user's main program

The following is a skeleton of the user's main program (or calling program) as an application of IDAS. The user program is to have these steps in the order indicated, unless otherwise noted. For the sake of brevity, we defer many of the details to the later sections. As in §5.1.3, most steps are independent of the `N_Vector`, `SUNMatrix`, `SUNLinearSolver`, and `SUNNonlinearSolver` implementations used. For the steps that are not, refer to Chapters §6, §7, §8, §9 for the specific name of the function to be called or macro to be referenced.

First, note that no additional header files need be included for forward sensitivity analysis beyond those for IVP solution §5.1.3.

Differences from the user main program skeleton in §5.1.3 are bolded.

1. Initialize parallel or multi-threaded environment
2. Create the SUNDIALS context object
3. Set the vector of initial values
4. Create matrix object
5. Create linear solver object
6. Create nonlinear solver object
7. Create IDAS object
8. Initialize IDAS solver
9. Specify integration tolerances
10. Attach linear solver
11. Set linear solver optional inputs
12. Attach nonlinear solver
13. Set nonlinear solver optional inputs
14. **Initialize quadrature integration**

    If the quadrature is not sensitivity-dependent, initialize the quadrature integration as described in §5.2. For integrating a problem where the quadrature depends on the forward sensitivities see §5.4.4.

15. **Set the sensitivity initial values**

    Call *N_VCloneVectorArray()* to create `N_Vector` arrays `yS0` and `ypS0` to hold the initial values for the sensitivity vectors of y and sensitivity derivative vectors of $\dot{y}$, respectively.

    ```
    yS0  = N_VCloneVectorArray(Ns, y0);
    ypS0 = N_VCloneVectorArray(Ns, y0);
    ```

    where `Ns` is the number of parameters with respect to which sensitivities are to be computed and `y0` serves only to provide an `N_Vector` template for cloning.

    Then, load initial values for each sensitivity vector `yS0[i]` and sensitivity derivative vector `ypS0[i]` for `i = 0,...,N_s-1`.

16. **Activate sensitivity calculations**

    Call *IDASensInit()* to activate forward sensitivity computations and allocate internal memory for IDAS related to sensitivity calculations.

    If a sensitivity residual function is *not* provided to *IDASensInit()*, then *IDASetSensParams() must* be called after *IDASensInit()* and before *IDASolve()* to provide the array of problem parameters with respect to which the sensitivities are computed. This array must also be attached to the "user data" pointer set with *IDASetUserData()*. Optionally, an array of scaling factors for difference-quotient residual computations and a mask array to select which parameters with respect to which the sensitivities are computed may also be provided to *IDASetSensParams()*.

    check *IDASetErrFile()*

17. **Set sensitivity integration tolerances (optional)**

    Call *IDASensSStolerances()* or *IDASensSVtolerances()* to set the sensitivity integration tolerances or *IDASensEEtolerances()* to have IDAS estimate tolerances for sensitivity variables based on the tolerances supplied for states variables.

    If sensitivity tolerances are estimated by IDAS, the results will be more accurate if order of magnitude is provided by setting the `pbar` input to *IDASetSensParams()*.

18. **Create sensitivity nonlinear solver**

    If using a non-default nonlinear solver (see §5.4.2.3), then create the desired nonlinear solver object by calling the appropriate constructor function defined by the particular `SUNNonlinearSolver` implementation e.g.,

    ```
    NLSSens = SUNNonlinSol_***Sens(...);
    ```

    for the `IDA_SIMULTANEOUS` or `IDA_STAGGERED` options `***` is the name of the nonlinear solver and `...` are constructor specific arguments (see §9 for details).

19. **Attach the sensitivity nonlinear solver**

    If using a non-default nonlinear solver, then initialize the nonlinear solver interface by attaching the nonlinear solver object by calling *IDASetNonlinearSolverSensSim()* when using the `IDA_SIMULTANEOUS` corrector method, *IDASetNonlinearSolverSensStg()* when using the `IDA_STAGGERED` corrector method (see §5.4.2.3 for details).

20. **Set sensitivity nonlinear solver optional inputs**

    Call the appropriate set functions for the selected nonlinear solver module to change optional inputs specific to that nonlinear solver. These *must* be called after *IDASensInit()* if using the default nonlinear solver or after attaching a new nonlinear solver to IDAS, otherwise the optional inputs will be overridden by IDAS defaults. See §9 for more information on optional inputs.

21. Specify rootfinding problem

22. **Set optional inputs**

    Call `IDASetSens*` routines to change from their default values any optional inputs that control the behavior of IDAS in computing forward sensitivities. See §5.4.2.7 for details.

23. Correct initial values

24. Advance solution in time

25. **Extract sensitivity solution**

    After each successful return from *IDASolve()*, the solution of the original IVP is available in the `y` argument of *IDASolve()*, while the sensitivity solution can be extracted into `yS` and `ypS` (which can be the same as `yS0` and `ypS0`) by calling one of the routines *IDAGetSens()*, *IDAGetSens1()*, *IDAGetSensDky()*, or *IDAGetSensDky1()*.

26. Get optional outputs

27. Deallocate memory

    Upon completion of the integration, deallocate memory for the vectors `yS0` and `yps0` using *N_VDestroyVectorArray()*.

28. Finalize MPI, if used

5.4.2 User-callable routines for forward sensitivity analysis

This section describes the IDAS functions, in addition to those presented in §5.1.4, that are called by the user to setup and solve a forward sensitivity problem.

5.4.2.1 Forward sensitivity initialization and deallocation functions

Activation of forward sensitivity computation is done by calling *IDASensInit()* or `IDASensInit1()`, depending on whether the sensitivity residual function returns all sensitivities at once or one by one, respectively. The form of the call to each of these routines is as follows:

int **IDASensInit**(void *ida_mem, int Ns, int ism, *IDASensResFn* fS, *N_Vector* *yS0, *N_Vector* *ypS0)

The routine *IDASensInit()* activates forward sensitivity computations and allocates internal memory related to sensitivity calculations.

Arguments:

- `ida_mem` – pointer to the IDAS memory block returned by *IDACreate()*.
- `Ns` – the number of sensitivities to be computed.
- `ism` – forward sensitivity analysis!correction strategies a flag used to select the sensitivity solution method. Its value can be `IDA_SIMULTANEOUS` or `IDA_STAGGERED` :
  - In the `IDA_SIMULTANEOUS` approach, the state and sensitivity variables are corrected at the same time. If the default Newton nonlinear solver is used, this amounts to performing a modified Newton iteration on the combined nonlinear system.
  - In the `IDA_STAGGERED` approach, the correction step for the sensitivity variables takes place at the same time for all sensitivity equations, but only after the correction of the state variables has converged and the state variables have passed the local error test.
- `resS` – is the C function which computes all sensitivity ODE residuals at the same time. For full details see *IDASensResFn*.
- `yS0` – a pointer to an array of `Ns` vectors containing the initial values of the sensitivities of y.

- ypS0 – a pointer to an array of Ns vectors containing the initial values of the sensitivities of $\dot{y}$.

Return value:

- IDA_SUCCESS – The call to *IDASensInit()* was successful.
- IDA_MEM_NULL – The IDAS memory block was not initialized through a previous call to *IDACreate()*.
- IDA_MEM_FAIL – A memory allocation request has failed.
- IDA_ILL_INPUT – An input argument to *IDASensInit()* has an illegal value.

Notes:

Passing fs == NULL indicates using the default internal difference quotient sensitivity residual routine and *IDASetSensParams()* *must* be called before *IDASolve()*.

If an error occurred, *IDASensInit()* also sends an error message to the error handler function.

In terms of the problem size N, number of sensitivity vectors N_s, and maximum method order maxord, the size of the real workspace is increased as follows:

- Base value: lenrw $=$ lenrw $+ (\texttt{maxord} + 5)N_sN$
- With *IDASensSVtolerances()*: $textttlenrw =$ lenrw $+ N_sN$

the size of the integer workspace is increased as follows:

- Base value: leniw $=$ leniw $+ (\texttt{maxord} + 5)N_sN_i$
- With *IDASensSVtolerances()*: leniw $=$ leniw $+ N_sN_i$

where N_i is the number of integers in one N_Vector.

The routine *IDASensReInit()*, useful during the solution of a sequence of problems of same size, reinitializes the sensitivity-related internal memory. The call to it must follow a call to *IDASensInit()* (and maybe a call to *IDAReInit()*). The number Ns of sensitivities is assumed to be unchanged since the call to the initialization function. The call to the *IDASensReInit()* function has the form:

int **IDASensReInit**(void *ida_mem, int ism, *N_Vector* *yS0, *N_Vector* *ypS0)

The routine *IDASensReInit()* reinitializes forward sensitivity computations.

Arguments:

- ida_mem – pointer to the IDAS memory block returned by *IDACreate()*.
- ism – forward sensitivity analysis!correction strategies a flag used to select the sensitivity solution method. Its value can be IDA_SIMULTANEOUS , IDA_STAGGERED , or IDA_STAGGERED1.
- yS0 – a pointer to an array of Ns variables of type N_Vector containing the initial values of the sensitivities.
- ypS0 – a pointer to an array of Ns variables of type N_Vector containing the initial values of the sensitivities of $\dot{y}$.

Return value:

- IDA_SUCCESS – The call to *IDASensReInit()* was successful.
- IDA_MEM_NULL – The IDAS memory block was not initialized through a previous call to *IDACreate()*.
- IDA_NO_SENS – Memory space for sensitivity integration was not allocated through a previous call to *IDASensInit()*.
- IDA_ILL_INPUT – An input argument to *IDASensReInit()* has an illegal value.

- IDA_MEM_FAIL – A memory allocation request has failed.

Notes:

All arguments of *IDASensReInit()* are the same as those of the functions *IDASensInit()*. If an error occurred, *IDASensReInit()* also sends a message to the error handler function.

To deallocate all forward sensitivity-related memory (allocated in a prior call to *IDASensInit()*), the user must call

void **IDASensFree**(void *ida_mem)

The function *IDASensFree()* frees the memory allocated for forward sensitivity computations by a previous call to *IDASensInit()*.

Arguments:

- ida_mem – pointer to the IDAS memory block returned by *IDACreate()*.

Return value:

- The function has no return value.

Notes: In general, *IDASensFree()* need not be called by the user, as it is invoked automatically by *IDAFree()*.

After a call to *IDASensFree()*, forward sensitivity computations can be reactivated only by calling *IDASensInit()*.

To activate and deactivate forward sensitivity calculations for successive IDAS runs, without having to allocate and deallocate memory, the following function is provided:

int **IDASensToggleOff**(void *ida_mem)

The function *IDASensToggleOff()* deactivates forward sensitivity calculations. It does not deallocate sensitivity-related memory.

Arguments:

- ida_mem – pointer to the memory previously returned by *IDACreate()*.

Return value:

- IDA_SUCCESS – *IDASensToggleOff()* was successful.
- IDA_MEM_NULL – ida_mem was NULL.

Notes: Since sensitivity-related memory is not deallocated, sensitivities can be reactivated at a later time (using *IDASensReInit()*).

5.4.2.2 Forward sensitivity tolerance specification functions

One of the following three functions must be called to specify the integration tolerances for sensitivities. Note that this call must be made after the call to *IDASensInit()*.

int **IDASensSStolerances**(void *ida_mem, *realtype* reltolS, *realtype* *abstolS)

The function *IDASensSStolerances()* specifies scalar relative and absolute tolerances.

Arguments:

- ida_mem – pointer to the IDAS memory block returned by *IDACreate()*.
- reltolS – is the scalar relative error tolerance.
- abstolS – is a pointer to an array of length Ns containing the scalar absolute error tolerances, one for each parameter.

Return value:

- IDA_SUCCESS – The call to *IDASStolerances()* was successful.

- IDA_MEM_NULL – The IDAS memory block was not initialized through a previous call to *IDACreate()*.
- IDA_NO_SENS – The sensitivity allocation function *IDASensInit()* has not been called.
- IDA_ILL_INPUT – One of the input tolerances was negative.

int **IDASensSVtolerances**(void *ida_mem, *realtype* reltolS, *N_Vector* *abstolS)

The function *IDASensSVtolerances()* specifies scalar relative tolerance and vector absolute tolerances.

Arguments:

- ida_mem – pointer to the IDAS memory block returned by *IDACreate()*.
- reltolS – is the scalar relative error tolerance.
- abstolS – is an array of Ns variables of type N_Vector. The N_Vector from abstolS[is] specifies the vector tolerances for is -th sensitivity.

Return value:

- IDA_SUCCESS – The call to *IDASVtolerances()* was successful.
- IDA_MEM_NULL – The IDAS memory block was not initialized through a previous call to *IDACreate()*.
- IDA_NO_SENS – The allocation function for sensitivities has not been called.
- IDA_ILL_INPUT – The relative error tolerance was negative or an absolute tolerance vector had a negative component.

Notes: This choice of tolerances is important when the absolute error tolerance needs to be different for each component of any vector yS[i].

int **IDASensEEtolerances**(void *ida_mem)

When *IDASensEEtolerances()* is called, IDAS will estimate tolerances for sensitivity variables based on the tolerances supplied for states variables and the scaling factors $\bar{p}$.

Arguments:

- ida_mem – pointer to the IDAS memory block returned by *IDACreate()*.

Return value:

- IDA_SUCCESS – The call to *IDASensEEtolerances()* was successful.
- IDA_MEM_NULL – The IDAS memory block was not initialized through a previous call to *IDACreate()*.
- IDA_NO_SENS – The sensitivity allocation function has not been called.

5.4.2.3 Forward sensitivity nonlinear solver interface functions

As in the pure DAE case, when computing solution sensitivities using forward sensitivitiy analysis IDAS uses the SUNNonlinearSolver implementation of Newton's method defined by the SUNNONLINSOL_NEWTON module (see §9.3) by default. To specify a different nonlinear solver in IDAS, the user's program must create a SUNNonlinearSolver object by calling the appropriate constructor routine. The user must then attach the SUNNonlinearSolver object to IDAS by calling *IDASetNonlinearSolverSensSim()* when using the IDA_SIMULTANEOUS corrector option, or *IDASetNonlinearSolver()* and *IDASetNonlinearSolverSensStg()* or IDASetNonlinearSolverSensStg1() when using the IDA_STAGGERED as documented below.

When changing the nonlinear solver in IDAS, *IDASetNonlinearSolver()* must be called after *IDAInit()*; similarly *IDASetNonlinearSolverSensSim()*, IDASetNonlinearSolverStg(), must be called after *IDASensInit()*. If

any calls to *IDASolve()* have been made, then IDAS will need to be reinitialized by calling *IDAReInit()* to ensure that the nonlinear solver is initialized correctly before any subsequent calls to *IDASolve()*.

The first argument passed to the routines *IDASetNonlinearSolverSensSim()*, and *IDASetNonlinearSolverSensStg()*, is the IDAS memory pointer returned by *IDACreate()* and the second argument is the SUNNonlinearSolver object to use for solving the nonlinear systems (2.4). A call to this function attaches the nonlinear solver to the main IDAS integrator.

int **IDASetNonlinearSolverSensSim**(void *ida_mem, *SUNNonlinearSolver* NLS)

The function IDASetNonLinearSolverSensSim() attaches a SUNNonlinearSolver object (NLS) to IDAS when using the IDA_SIMULTANEOUS approach to correct the state and sensitivity variables at the same time.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- NLS – SUNNonlinearSolver object to use for solving nonlinear system (2.4).

Return value:

- IDA_SUCCESS – The nonlinear solver was successfully attached.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_ILL_INPUT – The SUNNONLINSOL object is NULL , does not implement the required nonlinear solver operations, is not of the correct type, or the residual function, convergence test function, or maximum number of nonlinear iterations could not be set.

int **IDASetNonlinearSolverSensStg**(void *ida_mem, *SUNNonlinearSolver* NLS)

The function IDASetNonLinearSolverSensStg() attaches a SUNNonlinearSolver object (NLS) to IDAS when using the IDA_STAGGERED approach to correct all the sensitivity variables after the correction of the state variables.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- NLS – SUNNONLINSOL object to use for solving nonlinear systems.

Return value:

- IDA_SUCCESS – The nonlinear solver was successfully attached.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_ILL_INPUT – The SUNNONLINSOL object is NULL , does not implement the required nonlinear solver operations, is not of the correct type, or the residual function, convergence test function, or maximum number of nonlinear iterations could not be set.

Notes: This function only attaches the SUNNonlinearSolver object for correcting the sensitivity variables. To attach a SUNNonlinearSolver object for the state variable correction use *IDASetNonlinearSolver()*.

5.4.2.4 Forward sensitivity initial condition calculation function

IDACalcIC() also calculates corrected initial conditions for sensitivity variables of a DAE system. When used for initial conditions calculation of the forward sensitivities, *IDACalcIC()* must be preceded by successful calls to *IDASensInit()* (or *IDASensReInit()*) and should precede the call(s) to *IDASolve()*. For restrictions that apply for initial conditions calculation of the state variables, see §5.1.4.7.

Calling *IDACalcIC()* is optional. It is only necessary when the initial conditions do not satisfy the sensitivity systems. Even if forward sensitivity analysis was enabled, the call to the initial conditions calculation function *IDACalcIC()* is exactly the same as for state variables.

```
flag = IDACalcIC(ida_mem, icopt, tout1);
```

See *IDACalcIC()* for a list of possible return values.

5.4.2.5 IDAS solver function

Even if forward sensitivity analysis was enabled, the call to the main solver function *IDASolve()* is exactly the same as in §5.1. However, in this case the return value `flag` can also be one of the following:

- `IDA_SRES_FAIL` – The sensitivity residual function failed in an unrecoverable manner.
- `IDA_REP_SRES_ERR` – The user's residual function repeatedly returned a recoverable error flag, but the solver was unable to recover.

5.4.2.6 Forward sensitivity extraction functions

If forward sensitivity computations have been initialized by a call to *IDASensInit()*, or reinitialized by a call to *IDASensReInit()*, then IDAS computes both a solution and sensitivities at time `t`. However, *IDASolve()* will still return only the solution y in `yout`. Solution sensitivities can be obtained through one of the following functions:

int **IDAGetSens**(void *ida_mem, *realtype* *tret, *N_Vector* *yS)
The function *IDAGetSens()* returns the sensitivity solution vectors after a successful return from *IDASolve()*.

Arguments:

- `ida_mem` – pointer to the memory previously allocated by *IDAInit()*.
- `tret` – the time reached by the solver output.
- `yS` – array of computed forward sensitivity vectors. This vector array must be allocated by the user.

Return value:

- `IDA_SUCCESS` – *IDAGetSens()* was successful.
- `IDA_MEM_NULL` – `ida_mem` was `NULL`.
- `IDA_NO_SENS` – Forward sensitivity analysis was not initialized.
- `IDA_BAD_DKY` – `yS` is `NULL`.

Notes: Note that the argument `tret` is an output for this function. Its value will be the same as that returned at the last *IDASolve()* call.

The function *IDAGetSensDky()* computes the `k`-th derivatives of the interpolating polynomials for the sensitivity variables at time `t`. This function is called by *IDAGetSens()* with $\mathtt{k} = 0$, but may also be called directly by the user.

int **IDAGetSensDky**(void *ida_mem, *realtype* t, int k, *N_Vector* *dkyS)
The function *IDAGetSensDky()* returns derivatives of the sensitivity solution vectors after a successful return from *IDASolve()*.

Arguments:

- `ida_mem` – pointer to the memory previously allocated by *IDAInit()*.
- `t` – specifies the time at which sensitivity information is requested. The time `t` must fall within the interval defined by the last successful step taken by IDAS.
- `k` – order of derivatives. `k` must be in the range $0, 1, ..., klast$ where $klast$ is the method order of the last successful step.

- dkyS – array of Ns vectors containing the derivatives on output. The space for dkyS must be allocated by the user.

Return value:

- IDA_SUCCESS – *IDAGetSensDky()* succeeded.
- IDA_MEM_NULL – ida_mem was NULL.
- IDA_NO_SENS – Forward sensitivity analysis was not initialized.
- IDA_BAD_DKY – One of the vectors dkyS[i] is NULL.
- IDA_BAD_K – k is not in the range $0, 1, ..., \texttt{qlast}$.
- IDA_BAD_T – The time t is not in the allowed range.

Forward sensitivity solution vectors can also be extracted separately for each parameter in turn through the functions *IDAGetSens1()* and *IDAGetSensDky1()*, defined as follows:

int **IDAGetSens1**(void *ida_mem, *realtype* *tret, int is, *N_Vector* yS)

The function IDAGetSens1 returns the is-th sensitivity solution vector after a successful return from *IDASolve()*.

Arguments:

- ida_mem – pointer to the memory previously allocated by *IDAInit()*.
- tret – the time reached by the solver output.
- is – specifies which sensitivity vector is to be returned $0 \leq \texttt{is} < N_s$.
- yS – the computed forward sensitivity vector. This vector array must be allocated by the user.

Return value:

- IDA_SUCCESS – IDAGetSens1 was successful.
- IDA_MEM_NULL – ida_mem was NULL.
- IDA_NO_SENS – Forward sensitivity analysis was not initialized.
- IDA_BAD_IS – The index is is not in the allowed range.
- IDA_BAD_DKY – yS is NULL.
- IDA_BAD_T – The time t is not in the allowed range.

Notes: Note that the argument tret is an output for this function. Its value will be the same as that returned at the last *IDASolve()* call.

int **IDAGetSensDky1**(void *ida_mem, *realtype* t, int k, int is, *N_Vector* dkyS)

The function IDAGetSensDky1 returns the k-th derivative of the is-th sensitivity solution vector after a successful return from *IDASolve()*.

Arguments:

- ida_mem – pointer to the memory previously allocated by *IDAInit()*.
- t – specifies the time at which sensitivity information is requested. The time t must fall within the interval defined by the last successful step taken by IDAS.
- k – order of derivative.
- is – specifies the sensitivity derivative vector to be returned $0 \leq \texttt{is} < N_s$.
- dkyS – the vector containing the derivative. The space for dkyS must be allocated by the user.

Return value:

- IDA_SUCCESS – IDAGetQuadDky1 succeeded.
- IDA_MEM_NULL – The pointer to ida_mem was NULL.
- IDA_NO_SENS – Forward sensitivity analysis was not initialized.
- IDA_BAD_DKY – dkyS or one of the vectors dkyS[i] is NULL.
- IDA_BAD_IS – The index is is not in the allowed range.
- IDA_BAD_K – k is not in the range $0, 1, ..., qlast$.
- IDA_BAD_T – The time t is not in the allowed range.

5.4.2.7 Optional inputs for forward sensitivity analysis

Optional input variables that control the computation of sensitivities can be changed from their default values through calls to IDASetSens* functions. Table 5.7 lists all forward sensitivity optional input functions in IDAS which are described in detail in the remainder of this section.

We note that, on an error return, all of the optional input functions send an error message to the error handler function. All error return values are negative, so the test flag < 0 will catch all errors. Finally, a call to a IDASetSens*** function can be made from the user's calling program at any time and, if successful, takes effect immediately.

Table 5.7: Forward sensitivity optional inputs :align: center

Optional input	Routine name	Default
Sensitivity scaling factors	*IDASetSensParams()*	NULL
DQ approximation method	*IDASetSensDQMethod()*	centered/0.0
Error control strategy	*IDASetSensErrCon()*	SUNFALSE
Maximum no. of nonlinear iterations	*IDASetSensMaxNonlinIters()*	4

int **IDASetSensParams**(void *ida_mem, *realtype* *p, *realtype* *pbar, int *plist)

The function *IDASetSensParams()* specifies problem parameter information for sensitivity calculations.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- p – a pointer to the array of real problem parameters used to evaluate $F(t, y, \dot{y}, p)$. If non- NULL , p must point to a field in the user's data structure user_data passed to the residual function.
- pbar – an array of Ns positive scaling factors. If non- NULL , pbar must have all its components > 0.0.
- plist – an array of Ns non-negative indices to specify which components p[i] to use in estimating the sensitivity equations. If non- NULL , plist must have all components ≥ 0.

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_NO_SENS – Forward sensitivity analysis was not initialized.
- IDA_ILL_INPUT – An argument has an illegal value.

Note: The array p only needs to include the parameters with respect to which sensitivities are (potentially) desired.

If the user provides a function to evaluate the sensitivity residuals, p need not be specified.

When computing the sensitivity residual via a difference-quotient or estimating sensitivity tolerances the results will be more accurate if order of magnitude information is provided with `pbar`. Typically, if `p[0] != 0`, the value `pbar[i] = abs(p[plist[i]])` can be used. By default IDAS uses `p[i] = 1.0`.

If the user provides a function to evaluate the sensitivity residual and specifies tolerances for the sensitivity variables, `pbar` need not be specified.

By default IDA computes sensitivities with respect to the first `Ns` parameters in `p` i.e., `plist[i] = i` for `i = 0,...,Ns-1`. If sensitivities with respect to the j-th parameter `p[j]` are desired, set `plist[i] = j` for some $0 \leq i < N_s$ and $0 \leq j < N_p$ where N_p is the number of element in `p`.

If the user provides a function to evaluate the sensitivity residuals, `plist` need not be specified.

Warning: This function must be preceded by a call to *IDASensInit()*.

The array `p` *must* also be attached to the user data structure. For example, `user_data->p = p;`.

int **IDASetSensDQMethod**(void *ida_mem, int DQtype, *realtype* DQrhomax)

The function *IDASetSensDQMethod()* specifies the difference quotient strategy in the case in which the residual of the sensitivity equations are to be computed by IDAS.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `DQtype` – specifies the difference quotient type. Its value can be `IDA_CENTERED` or `IDA_FORWARD`.
- `DQrhomax` – positive value of the selection parameter used in deciding switching between a simultaneous or separate approximation of the two terms in the sensitivity residual.

Return value:

- `IDA_SUCCESS` – The optional value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDA_ILL_INPUT` – An argument has an illegal value.

Notes:

If `DQrhomax` $= 0.0$, then no switching is performed. The approximation is done simultaneously using either centered or forward finite differences, depending on the value of `DQtype`. For values of `DQrhomax` ≥ 1.0, the simultaneous approximation is used whenever the estimated finite difference perturbations for states and parameters are within a factor of `DQrhomax`, and the separate approximation is used otherwise. Note that a value `DQrhomax` < 1.0 will effectively disable switching. See §2.5 for more details.

The default value are `DQtype == IDA_CENTERED` and `DQrhomax`$= 0.0$.

int **IDASetSensErrCon**(void *ida_mem, *booleantype* errconS)

The function *IDASetSensErrCon()* specifies the error control strategy for sensitivity variables.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `errconS` – specifies whether sensitivity variables are to be included `SUNTRUE` or not `SUNFALSE` in the error control mechanism.

Return value:

- `IDA_SUCCESS` – The optional value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.

Notes: By default, errconS is set to SUNFALSE. If errconS = SUNTRUE then both state variables and sensitivity variables are included in the error tests. If errconS = SUNFALSE then the sensitivity variables are excluded from the error tests. Note that, in any event, all variables are considered in the convergence tests.

int **IDASetSensMaxNonlinIters**(void *ida_mem, int maxcorS)

The function *IDASetSensMaxNonlinIters()* specifies the maximum number of nonlinear solver iterations for sensitivity variables per step.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- maxcorS – maximum number of nonlinear solver iterations allowed per step > 0.

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_MEM_FAIL – The SUNNONLINSOL module is NULL.

Notes: The default value is 3.

5.4.2.8 Optional outputs for forward sensitivity analysis

Optional output functions that return statistics and solver performance information related to forward sensitivity computations are listed in Table 5.8 and described in detail in the remainder of this section.

Table 5.8: Forward sensitivity optional outputs

Optional output	Routine name
No. of calls to sensitivity residual function	*IDAGetSensNumResEvals()*
No. of calls to residual function for sensitivity	*IDAGetNumResEvalsSens()*
No. of sensitivity local error test failures	*IDAGetSensNumErrTestFails()*
No. of calls to lin. solv. setup routine for sens.	*IDAGetSensNumLinSolvSetups()*
Error weight vector for sensitivity variables	*IDAGetSensErrWeights()*
Sensitivity-related statistics as a group	*IDAGetSensStats()*
No. of sens. nonlinear solver iterations	*IDAGetSensNumNonlinSolvIters()*
No. of sens. convergence failures	*IDAGetSensNumNonlinSolvConvFails()*
Sens. nonlinear solver statistics as a group	IDAGetSensNonlinSolveStats()

int **IDAGetSensNumResEvals**(void *ida_mem, long int *nfSevals)

The function *IDAGetSensNumResEvals()* returns the number of calls to the sensitivity residual function.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- nfSevals – number of calls to the sensitivity residual function.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_NO_SENS – Forward sensitivity analysis was not initialized.

int **IDAGetNumResEvalsSens**(void *ida_mem, long int *nfevalsS)

The function IDAGetNumResEvalsSEns() returns the number of calls to the user's residual function due to the internal finite difference approximation of the sensitivity residuals.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- nfevalsS – number of calls to the user's DAE residual function for the evaluation of sensitivity residuals.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_NO_SENS – Forward sensitivity analysis was not initialized.

Notes: This counter is incremented only if the internal finite difference approximation routines are used for the evaluation of the sensitivity residuals.

int **IDAGetSensNumErrTestFails**(void *ida_mem, long int *nSetfails)

The function *IDAGetSensNumErrTestFails()* returns the number of local error test failures for the sensitivity variables that have occurred.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- nSetfails – number of error test failures.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_NO_SENS – Forward sensitivity analysis was not initialized.

Notes: This counter is incremented only if the sensitivity variables have been included in the error test (see *IDASetSensErrCon()*). Even in that case, this counter is not incremented if the ism = IDA_SIMULTANEOUS sensitivity solution method has been used.

int **IDAGetSensNumLinSolvSetups**(void *ida_mem, long int *nlinsetupsS)

The function *IDAGetSensNumLinSolvSetups()* returns the number of calls to the linear solver setup function due to forward sensitivity calculations.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- nlinsetupsS – number of calls to the linear solver setup function.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_NO_SENS – Forward sensitivity analysis was not initialized.

Notes: This counter is incremented only if a nonlinear solver requiring a linear solve has been used and the ism = IDA_STAGGERED sensitivity solution method has been specified (see §5.4.2.1).

int **IDAGetSensStats**(void *ida_mem, long int *nresSevals, long int *nresevalsS, long int *nSetfails, long int *nlinsetupsS)

The function *IDAGetSensStats()* returns all of the above sensitivity-related solver statistics as a group.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `nresSevals` – number of calls to the sensitivity residual function.
- `nresevalsS` – number of calls to the user-supplied DAE residual function for sensitivity evaluations.
- `nSetfails` – number of error test failures.
- `nlinsetupsS` – number of calls to the linear solver setup function.

Return value:

- `IDA_SUCCESS` – The optional output values have been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDA_NO_SENS` – Forward sensitivity analysis was not initialized.

int **IDAGetSensErrWeights**(void *ida_mem, *N_Vector* *eSweight)

The function *IDAGetSensErrWeights()* returns the sensitivity error weight vectors at the current time. These are the reciprocals of the W_i of (2.7) for the sensitivity variables.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `eSweight` – pointer to the array of error weight vectors.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDA_NO_SENS` – Forward sensitivity analysis was not initialized.

Notes: The user must allocate memory for `eweightS`.

int **IDAGetSensNumNonlinSolvIters**(void *ida_mem, long int *nSniters)

The function *IDAGetSensNumNonlinSolvIters()* returns the number of nonlinear iterations performed for sensitivity calculations.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `nSniters` – number of nonlinear iterations performed.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDA_NO_SENS` – Forward sensitivity analysis was not initialized.
- `IDA_MEM_FAIL` – The SUNNONLINSOL module is `NULL`.

Notes: This counter is incremented only if `ism` was `IDA_STAGGERED` or in the call to *IDASensInit()*.

int **IDAGetSensNumNonlinSolvConvFails**(void *ida_mem, long int *nSncfails)

The function *IDAGetSensNumNonlinSolvConvFails()* returns the number of nonlinear convergence failures that have occurred for sensitivity calculations.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- nSncfails – number of nonlinear convergence failures.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_NO_SENS – Forward sensitivity analysis was not initialized.

Notes: This counter is incremented only if ism was IDA_STAGGERED or in the call to *IDASensInit()*.

int **IDAGetSensNonlinSolvStats**(void *ida_mem, long int *nSniters, long int *nSncfails)
The function *IDAGetSensNonlinSolvStats()* returns the sensitivity-related nonlinear solver statistics as a group.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- nSniters – number of nonlinear iterations performed.
- nSncfails – number of nonlinear convergence failures.

Return value:

- IDA_SUCCESS – The optional output values have been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_NO_SENS – Forward sensitivity analysis was not initialized.
- IDA_MEM_FAIL – The SUNNONLINSOL module is NULL.

5.4.2.9 Initial condition calculation optional output functions

The sensitivity consistent initial conditions found by IDAS (after a successful call to *IDACalcIC()*) can be obtained by calling the following function:

int **IDAGetSensConsistentIC**(void *ida_mem, *N_Vector* *yyS0_mod, *N_Vector* *ypS0_mod)
The function *IDAGetSensConsistentIC()* returns the corrected initial conditions calculated by *IDACalcIC()* for sensitivities variables.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- yyS0_mod – a pointer to an array of Ns vectors containing consistent sensitivity vectors.
- ypS0_mod – a pointer to an array of Ns vectors containing consistent sensitivity derivative vectors.

Return value:

- IDA_SUCCESS – *IDAGetSensConsistentIC()* succeeded.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_NO_SENS – The function *IDASensInit()* has not been previously called.
- IDA_ILL_INPUT – *IDASolve()* has been already called.

Notes: If the consistent sensitivity vectors or consistent derivative vectors are not desired, pass `NULL` for the corresponding argument.

> **Warning:** The user must allocate space for `yyS0_mod` and `ypS0_mod` (if not `NULL`).

5.4.3 User-supplied routines for forward sensitivity analysis

In addition to the required and optional user-supplied routines described in §5.1.5, when using IDAS for forward sensitivity analysis, the user has the option of providing a routine that calculates the residual of the sensitivity equations (2.11).

By default, IDAS uses difference quotient approximation routines for the residual of the sensitivity equations. However, IDAS allows the option for user-defined sensitivity residual routines (which also provides a mechanism for interfacing IDAS to routines generated by automatic differentiation).

The user may provide the residuals of the sensitivity equations (2.11) for all sensitivity parameters at once, through a function of type *IDASensResFn* defined by:

typedef int (***IDASensResFn**)(int Ns, *realtype* t, *N_Vector* yy, *N_Vector* yp, *N_Vector* resval, *N_Vector* *yS, *N_Vector* *ypS, *N_Vector* *resvalS, void *user_data, *N_Vector* tmp1, *N_Vector* tmp2, *N_Vector* tmp3)

This function computes the sensitivity residual for all sensitivity equations. It must compute the vectors $(\partial F/\partial y_i)\, s_i(t) + (\partial F/\partial \dot{y})\, \dot{s}_i(t) + (\partial F/\partial p_i)$ and store them in `resvalS[i]`.

Arguments:

- `Ns` – is the number of sensitivities.
- `t` – is the current value of the independent variable.
- `yy` – is the current value of the state vector, $y(t)$.
- `yp` – is the current value of $\dot{y}(t)$.
- `resval` – contains the current value F of the original DAE residual.
- `yS` – contains the current values of the sensitivities s_i .
- `ypS` – contains the current values of the sensitivity derivatives $\dot{s}_i$.
- `resvalS` – contains the output sensitivity residual vectors. Memory allocation for `resvalS` is handled within IDAS.
- `user_data` – is a pointer to user data.
- `tmp1`, `tmp2`, `tmp3` – are `N_Vector` s of length N which can be used as temporary storage.

Return value: An *IDASensResFn()* should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `IDA_SRES_FAIL` is returned).

Notes: There is one situation in which recovery is not possible even if *IDASensResFn()* function returns a recoverable error flag. That is when this occurs at the very first call to the *IDASensResFn()*, in which case IDAS returns `IDA_FIRST_RES_FAIL`.

5.4.4 Integration of quadrature equations depending on forward sensitivities

IDAS provides support for integration of quadrature equations that depends not only on the state variables but also on forward sensitivities.

The following is an overview of the sequence of calls in a user's main program in this situation. Steps that are changed from the skeleton program presented in §5.1.3 are bolded. See also §5.2.

1. Initialize parallel or multi-threaded environment, if appropriate
2. Create the SUNDIALS context object
3. Set vector of initial values
4. Create matrix object
5. Create linear solver object
6. Set linear solver optional inputs
7. Create nonlinear solver object
8. Create IDAS object
9. Initialize IDAS solver
10. Specify integration tolerances
11. Attach linear solver
12. Set linear solver optional inputs
13. Attach nonlinear solver
14. Set nonlinear solver optional inputs
15. Set sensitivity initial values
16. Activate sensitivity calculations
17. Set sensitivity integration tolerances
18. Create sensitivity nonlinear solver
19. Attach the sensitivity nonlinear solver
20. Set sensitivity nonlinear solver optional inputs
21. **Set vector of initial values for quadrature variables**

    Typically, the quadrature variables should be initialized to 0.
22. **Initialize sensitivity-dependent quadrature integration**

    Call *IDAQuadSensInit()* to specify the quadrature equation right-hand side function and to allocate internal memory related to quadrature integration.
23. Specify rootfinding problem
24. **Set optional inputs**

    Call *IDASetQuadSensErrCon()* to indicate whether or not quadrature variables should be used in the step size control mechanism. If so, one of the `IDAQuadSens*tolerances` functions must be called to specify the integration tolerances for quadrature variables. See §5.2.4 for details.
25. Correct initial values
26. Advance solution in time

27. Extract sensitivity solution
28. **Extract sensitivity-dependent quadrature variables**

    Call *IDAGetQuadSens()*, *IDAGetQuadSens1()*, *IDAGetQuadSensDky()* or *IDAGetQuadSensDky1()* to obtain the values of the quadrature variables or their derivatives at the current time.
29. **Get optional outputs**

    Call `IDAGetQuadSens*` functions to obtain optional output related to the integration of sensitivity-dependent quadratures. See §5.4.4.5 for details.
30. Deallocate memory
31. Finalize MPI, if used

5.4.4.1 Sensitivity-dependent quadrature initialization and deallocation

The function *IDAQuadSensInit()* activates integration of quadrature equations depending on sensitivities and allocates internal memory related to these calculations. If `rhsQS` is input as `NULL`, then IDAS uses an internal function that computes difference quotient approximations to the functions $\bar{q}_i = (\partial q/\partial y)s_i + (\partial q/\partial \dot{y})\dot{s}_i + \partial q/\partial p_i$, in the notation of (2.10). The form of the call to this function is as follows:

int **IDAQuadSensInit**(void *ida_mem, *IDAQuadSensRhsFn* rhsQS, *N_Vector* *yQS0)

The function *IDAQuadSensInit()* provides required problem specifications, allocates internal memory, and initializes quadrature integration.

Arguments:

- `ida_mem` – pointer to the IDAS memory block returned by *IDACreate()*.
- `rhsQS` – is the *IDAQuadSensRhsFn* function which computes f_{QS} , the right-hand side of the sensitivity-dependent quadrature equations.
- `yQS0` – contains the initial values of sensitivity-dependent quadratures.

Return value:

- `IDA_SUCCESS` – The call to *IDAQuadSensInit()* was successful.
- `IDA_MEM_NULL` – The IDAS memory was not initialized by a prior call to *IDACreate()*.
- `IDA_MEM_FAIL` – A memory allocation request failed.
- `IDA_NO_SENS` – The sensitivities were not initialized by a prior call to *IDASensInit()*.
- `IDA_ILL_INPUT` – The parameter `yQS0` is `NULL`.

Notes:

> **Warning:** Before calling *IDAQuadSensInit()*, the user must enable the sensitivites by calling *IDASensInit()*. If an error occurred, *IDAQuadSensInit()* also sends an error message to the error handler function.

In terms of the number of quadrature variables N_q and maximum method order `maxord`, the size of the real workspace is increased as follows:

- Base value: `lenrw` $=$ `lenrw` $+ (\texttt{maxord} + 5)N_q$
- If *IDAQuadSensSVtolerances()* is called: `lenrw` $=$ `lenrw` $+ N_q N_s$

and the size of the integer workspace is increased as follows:

- Base value: leniw = leniw + (maxord + 5)N_q
- If *IDAQuadSensSVtolerances()* is called: leniw = leniw + $N_q N_s$

The function *IDAQuadSensReInit()*, useful during the solution of a sequence of problems of same size, reinitializes the quadrature related internal memory and must follow a call to *IDAQuadSensInit()*. The number Nq of quadratures as well as the number Ns of sensitivities are assumed to be unchanged from the prior call to *IDAQuadSensInit()*. The call to the *IDAQuadSensReInit()* function has the form:

int **IDAQuadSensReInit**(void *ida_mem, *N_Vector* *yQS0)

The function *IDAQuadSensReInit()* provides required problem specifications and reinitializes the sensitivity-dependent quadrature integration.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- yQS0 – contains the initial values of sensitivity-dependent quadratures.

Return value:

- IDA_SUCCESS – The call to *IDAQuadSensReInit()* was successful.
- IDA_MEM_NULL – The IDAS memory was not initialized by a prior call to *IDACreate()*.
- IDA_NO_SENS – Memory space for the sensitivity calculation was not allocated by a prior call to *IDASensInit()*.
- IDA_NO_QUADSENS – Memory space for the sensitivity quadratures integration was not allocated by a prior call to *IDAQuadSensInit()*.
- IDA_ILL_INPUT – The parameter yQS0 is NULL.

Notes: If an error occurred, *IDAQuadSensReInit()* also sends an error message to the error handler function.

void **IDAQuadSensFree**(void *ida_mem);

The function *IDAQuadSensFree()* frees the memory allocated for sensitivity quadrature integration.

Arguments:

- ida_mem – pointer to the IDAS memory block.

Return value: There is no return value.

Notes: In general, *IDAQuadSensFree()* need not be called by the user as it is called automatically by *IDAFree()*.

5.4.4.2 IDAS solver function

Even if quadrature integration was enabled, the call to the main solver function *IDASolve()* is exactly the same as in §5.1. However, in this case the return value flag can also be one of the following:

- IDA_QSRHS_FAIL – the sensitivity quadrature right-hand side function failed in an unrecoverable manner.
- IDA_FIRST_QSRHS_ERR – the sensitivity quadrature right-hand side function failed at the first call.
- IDA_REP_QSRHS_ERR – convergence test failures occurred too many times due to repeated recoverable errors in the quadrature right-hand side function. The IDA_REP_RES_ERR will also be returned if the quadrature right-hand side function had repeated recoverable errors during the estimation of an initial step size (assuming the sensitivity quadrature variables are included in the error tests).

5.4.4.3 Sensitivity-dependent quadrature extraction functions

If sensitivity-dependent quadratures have been initialized by a call to *IDAQuadSensInit()*, or reinitialized by a call to *IDAQuadSensReInit()*, then IDAS computes a solution, sensitivities, and quadratures depending on sensitivities at time `t`. However, *IDASolve()* will still return only the solutions y and $\dot{y}$. Sensitivity-dependent quadratures can be obtained using one of the following functions:

int **IDAGetQuadSens**(void *ida_mem, *realtype* *tret, *N_Vector* *yQS)

The function *IDAGetQuadSens()* returns the quadrature sensitivity solution vectors after a successful return from *IDASolve()*.

Arguments:

- `ida_mem` – pointer to the memory previously allocated by *IDAInit()*.
- `tret` – the time reached by the solver output.
- `yQS` – array of `Ns` computed sensitivity-dependent quadrature vectors. This array of vectors must be allocated by the user.

Return value:

- `IDA_SUCCESS` – *IDAGetQuadSens()* was successful.
- `IDA_MEM_NULL` – `ida_mem` was NULL.
- `IDA_NO_SENS` – Sensitivities were not activated.
- `IDA_NO_QUADSENS` – Quadratures depending on the sensitivities were not activated.
- `IDA_BAD_DKY` – `yQS` or one of the `yQS[i]` is `NULL`.

The function *IDAGetQuadSensDky()* computes the `k`-th derivatives of the interpolating polynomials for the sensitivity-dependent quadrature variables at time `t`. This function is called by *IDAGetQuadSens()* with `k = 0`, but may also be called directly by the user.

int **IDAGetQuadSensDky**(void *ida_mem, *realtype* t, int k, *N_Vector* *dkyQS)

The function *IDAGetQuadSensDky()* returns derivatives of the quadrature sensitivities solution vectors after a successful return from *IDASolve()*.

Arguments:

- `ida_mem` – pointer to the memory previously allocated by *IDAInit()*.
- `t` – the time at which information is requested. The time `t` must fall within the interval defined by the last successful step taken by IDAS.
- `k` – order of the requested derivative. `k` must be in the range $0, 1, ..., klast$ where $klast$ is the method order of the last successful step.
- `dkyQS` – array of `Ns` vectors containing the derivatives. This vector array must be allocated by the user.

Return value:

- `IDA_SUCCESS` – *IDAGetQuadSensDky()* succeeded.
- `IDA_MEM_NULL` – `ida_mem` was `NULL`.
- `IDA_NO_SENS` – Sensitivities were not activated.
- `IDA_NO_QUADSENS` – Quadratures depending on the sensitivities were not activated.
- `IDA_BAD_DKY` – `dkyQS` or one of the vectors `dkyQS[i]` is `NULL`.
- `IDA_BAD_K` – `k` is not in the range $0, 1, ..., klast$.
- `IDA_BAD_T` – The time `t` is not in the allowed range.

Quadrature sensitivity solution vectors can also be extracted separately for each parameter in turn through the functions IDAGetQuadSens1 and IDAGetQuadSensDky1, defined as follows:

int **IDAGetQuadSens1**(void *ida_mem, *realtype* *tret, int is, *N_Vector* yQS)

The function IDAGetQuadSens1 returns the is-th sensitivity of quadratures after a successful return from *IDASolve()*.

Arguments:

- ida_mem – pointer to the memory previously allocated by *IDAInit()*.
- tret – the time reached by the solver output.
- is – specifies which sensitivity vector is to be returned $0 \leq$ is $< N_s$.
- yQS – the computed sensitivity-dependent quadrature vector. This vector must be allocated by the user.

Return value:

- IDA_SUCCESS – IDAGetQuadSens1 was successful.
- IDA_MEM_NULL – ida_mem was NULL.
- IDA_NO_SENS – Forward sensitivity analysis was not initialized.
- IDA_NO_QUADSENS – Quadratures depending on the sensitivities were not activated.
- IDA_BAD_IS – The index is is not in the allowed range.
- IDA_BAD_DKY – yQS is NULL.

int **IDAGetQuadSensDky1**(void *ida_mem, *realtype* t, int k, int is, *N_Vector* dkyQS)

The function IDAGetQuadSensDky1 returns the k-th derivative of the is-th sensitivity solution vector after a successful return from *IDASolve()*.

Arguments:

- ida_mem – pointer to the memory previously allocated by *IDAInit()*.
- t – specifies the time at which sensitivity information is requested. The time t must fall within the interval defined by the last successful step taken by IDAS.
- k – order of derivative. k must be in the range $0, 1, ..., klast$ where $klast$ is the method order of the last successful step.
- is – specifies the sensitivity derivative vector to be returned $0 \leq$ is $< N_s$.
- dkyQS – the vector containing the derivative. The space for dkyQS must be allocated by the user.

Return value:

- IDA_SUCCESS – IDAGetQuadDky1 succeeded.
- IDA_MEM_NULL – ida_mem was NULL.
- IDA_NO_SENS – Forward sensitivity analysis was not initialized.
- IDA_NO_QUADSENS – Quadratures depending on the sensitivities were not activated.
- IDA_BAD_DKY – dkyQS is NULL.
- IDA_BAD_IS – The index is is not in the allowed range.
- IDA_BAD_K – k is not in the range $0, 1, ..., klast$.
- IDA_BAD_T – The time t is not in the allowed range.

5.4.4.4 Optional inputs for sensitivity-dependent quadrature integration

IDAS provides the following optional input functions to control the integration of sensitivity-dependent quadrature equations.

int **IDASetQuadSensErrCon**(void *ida_mem, *booleantype* errconQS)
The function *IDASetQuadSensErrCon()* specifies whether or not the quadrature variables are to be used in the local error control mechanism. If they are, the user must specify the error tolerances for the quadrature variables by calling *IDAQuadSensSStolerances()*, *IDAQuadSensSVtolerances()*, or *IDAQuadSensEEtolerances()*.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `errconQS` – specifies whether sensitivity quadrature variables are included `SUNTRUE` or not `SUNFALSE` in the error control mechanism.

Return value:

- `IDA_SUCCESS` – The optional value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDA_NO_SENS` – Sensitivities were not activated.
- `IDA_NO_QUADSENS` – Quadratures depending on the sensitivities were not activated.

Notes: By default, `errconQS` is set to `SUNFALSE`.

> **Warning:** It is illegal to call *IDASetQuadSensErrCon()* before a call to *IDAQuadSensInit()*.

If the quadrature variables are part of the step size control mechanism, one of the following functions must be called to specify the integration tolerances for quadrature variables.

int **IDAQuadSensSStolerances**(void *ida_mem, *realtype* reltolQS, *realtype* *abstolQS)
The function *IDAQuadSensSStolerances()* specifies scalar relative and absolute tolerances.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `reltolQS` – tolerances is the scalar relative error tolerance.
- `abstolQS` – is a pointer to an array containing the `Ns` scalar absolute error tolerances.

Return value:

- `IDA_SUCCESS` – The optional value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDA_NO_SENS` – Sensitivities were not activated.
- `IDA_NO_QUADSENS` – Quadratures depending on the sensitivities were not activated.
- `IDA_ILL_INPUT` – One of the input tolerances was negative.

int **IDAQuadSensSVtolerances**(void *ida_mem, *realtype* reltolQS, *N_Vector* *abstolQS)
The function *IDAQuadSensSVtolerances()* specifies scalar relative and vector absolute tolerances.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.

- reltolQS – tolerances is the scalar relative error tolerance.
- abstolQS – is an array of Ns variables of type N_Vector. The N_Vector from abstolS[is] specifies the vector tolerances for is -th quadrature sensitivity.

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_NO_QUAD – Quadrature integration was not initialized.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_NO_SENS – Sensitivities were not activated.
- IDA_NO_QUADSENS – Quadratures depending on the sensitivities were not activated.
- IDA_ILL_INPUT – One of the input tolerances was negative.

int **IDAQuadSensEEtolerances**(void *ida_mem)

The function *IDAQuadSensEEtolerances()* specifies that the tolerances for the sensitivity-dependent quadratures should be estimated from those provided for the pure quadrature variables.

Arguments:

- ida_mem – pointer to the IDAS memory block.

Return value:

- IDA_SUCCESS – The optional value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_NO_SENS – Sensitivities were not activated.
- IDA_NO_QUADSENS – Quadratures depending on the sensitivities were not activated.

Notes: When *IDAQuadSensEEtolerances()* is used, before calling *IDASolve()*, integration of pure quadratures must be initialized (see §5.2) and tolerances for pure quadratures must be also specified (see §5.2.4).

5.4.4.5 Optional outputs for sensitivity-dependent quadrature integration

IDAS provides the following functions that can be used to obtain solver performance information related to quadrature integration.

int **IDAGetQuadSensNumRhsEvals**(void *ida_mem, long int *nrhsQSevals)

The function *IDAGetQuadSensNumRhsEvals()* returns the number of calls made to the user's quadrature right-hand side function.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- nrhsQSevals – number of calls made to the user's rhsQS function.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_NO_QUADSENS – Sensitivity-dependent quadrature integration has not been initialized.

int **IDAGetQuadSensNumErrTestFails**(void *ida_mem, long int *nQSetfails)

The function *IDAGetQuadSensNumErrTestFails()* returns the number of local error test failures due to quadrature variables.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- nQSetfails – number of error test failures due to quadrature variables.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_NO_QUADSENS – Sensitivity-dependent quadrature integration has not been initialized.

int **IDAGetQuadSensErrWeights**(void *ida_mem, *N_Vector* *eQSweight)

The function *IDAGetQuadSensErrWeights()* returns the quadrature error weights at the current time.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- eQSweight – array of quadrature error weight vectors at the current time.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.
- IDA_NO_QUADSENS – Sensitivity-dependent quadrature integration has not been initialized.

Notes:

> **Warning:** The user must allocate memory for eQSweight. If quadratures were not included in the error control mechanism (through a call to *IDASetQuadSensErrCon()* with errconQS=SUNTRUE), *IDAGetQuadSensErrWeights()* does not set the eQSweight vector.

int **IDAGetQuadSensStats**(void *ida_mem, long int *nrhsQSevals, long int *nQSetfails)

The function *IDAGetQuadSensStats()* returns the IDAS integrator statistics as a group.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- nrhsQSevals – number of calls to the user's rhsQS function.
- nQSetfails – number of error test failures due to quadrature variables.

Return value:

- IDA_SUCCESS – the optional output values have been successfully set.
- IDA_MEM_NULL – the ida_mem pointer is NULL.
- IDA_NO_QUADSENS – Sensitivity-dependent quadrature integration has not been initialized.

5.4.4.6 User-supplied function for sensitivity-dependent quadrature integration

For the integration of sensitivity-dependent quadrature equations, the user must provide a function that defines the residual of those quadrature equations. For the sensitivities of quadratures (2.10) with integrand q, the appropriate residual functions are given by $\bar{q}_i = \partial q/\partial y s_i + \partial q/\partial \dot{y} \dot{s}_i + \partial q \partial p_i$. This user function must be of type *IDAQuadSensRhsFn* defined as follows:

typedef int (***IDAQuadSensRhsFn**)(int Ns, *realtype* t, *N_Vector* yy, *N_Vector* yp, *N_Vector* *yyS, *N_Vector* *ypS, *N_Vector* rrQ, *N_Vector* *rhsvalQS, void *user_data, *N_Vector* tmp1, *N_Vector* tmp2, *N_Vector* tmp3)

This function computes the sensitivity quadrature equation right-hand side for a given value of the independent variable t and state vector y.

Arguments:

- `Ns` – is the number of sensitivity vectors.
- `t` – is the current value of the independent variable.
- `yy` – is the current value of the dependent variable vector, $y(t)$.
- `yp` – is the current value of the dependent variable vector, $\dot{y}(t)$.
- `yyS` – is an array of `Ns` variables of type `N_Vector` containing the dependent sensitivity vectors s_i.
- `ypS` – is an array of `Ns` variables of type `N_Vector` containing the dependent sensitivity derivatives $\dot{s}_i$.
- `rrQ` – is the current value of the quadrature right-hand side q.
- `rhsvalQS` – contains the `Ns` output vectors.
- `user_data` – is the `user_data` pointer passed to *IDASetUserData()*.
- `tmp1`, `tmp2`, `tmp3` – are `N_Vector` s which can be used as temporary storage.

Return value: An *IDAQuadSensRhsFn* should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `IDA_QRHS_FAIL` is returned).

Notes:

Allocation of memory for `rhsvalQS` is automatically handled within IDAS.

Both `yy` and `yp` are of type `N_Vector` and both `yyS` and `ypS` are pointers to an array containing `Ns` vectors of type `N_Vector`. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each `N_Vector` implementation).

There is one situation in which recovery is not possible even if *IDAQuadSensRhsFn* function returns a recoverable error flag. That is when this occurs at the very first call to the *IDAQuadSensRhsFn*, in which case IDAS returns `IDA_FIRST_QSRHS_ERR`).

5.4.5 Note on using partial error control

For some problems, when sensitivities are excluded from the error control test, the behavior of IDAS may appear at first glance to be erroneous. One would expect that, in such cases, the sensitivity variables would not influence in any way the step size selection.

The short explanation of this behavior is that the step size selection implemented by the error control mechanism in IDAS is based on the magnitude of the correction calculated by the nonlinear solver. As mentioned in §5.4.2.1, even with partial error control selected in the call to *IDASensInit()*, the sensitivity variables are included in the convergence tests of the nonlinear solver.

When using the simultaneous corrector method §2.5, the nonlinear system that is solved at each step involves both the state and sensitivity equations. In this case, it is easy to see how the sensitivity variables may affect the convergence rate of the nonlinear solver and therefore the step size selection. The case of the staggered corrector approach is more subtle. The sensitivity variables at a given step are computed only once the solver for the nonlinear state equations has converged. However, if the nonlinear system corresponding to the sensitivity equations has convergence problems, IDAS will attempt to improve the initial guess by reducing the step size in order to provide a better prediction of the sensitivity variables. Moreover, even if there are no convergence failures in the solution of the sensitivity system, IDAS may trigger a call to the linear solver's setup routine which typically involves reevaluation of Jacobian information (Jacobian approximation in the case of matrix-based linear solvers, or preconditioner data in the case of the Krylov solvers). The new Jacobian information will be used by subsequent calls to the nonlinear solver for the state equations and, in this way, potentially affect the step size selection.

When using the simultaneous corrector method it is not possible to decide whether nonlinear solver convergence failures or calls to the linear solver setup routine have been triggered by convergence problems due to the state or the sensitivity equations. When using one of the staggered corrector methods, however, these situations can be identified by carefully monitoring the diagnostic information provided through optional outputs. If there are no convergence failures in the sensitivity nonlinear solver, and none of the calls to the linear solver setup routine were made by the sensitivity nonlinear solver, then the step size selection is not affected by the sensitivity variables.

Finally, the user must be warned that the effect of appending sensitivity equations to a given system of DAEs on the step size selection (through the mechanisms described above) is problem-dependent and can therefore lead to either an increase or decrease of the total number of steps that IDAS takes to complete the simulation. At first glance, one would expect that the impact of the sensitivity variables, if any, would be in the direction of increasing the step size and therefore reducing the total number of steps. The argument for this is that the presence of the sensitivity variables in the convergence test of the nonlinear solver can only lead to additional iterations (and therefore a smaller iteration error), or to additional calls to the linear solver setup routine (and therefore more up-to-date Jacobian information), both of which will lead to larger steps being taken by IDAS. However, this is true only locally. Overall, a larger integration step taken at a given time may lead to step size reductions at later times, due to either nonlinear solver convergence failures or error test failures.

5.5 Using IDAS for Adjoint Sensitivity Analysis

This chapter describes the use of IDAS to compute sensitivities of derived functions using adjoint sensitivity analysis. As mentioned before, the adjoint sensitivity module of IDAS provides the infrastructure for integrating backward in time any system of DAEs that depends on the solution of the original IVP, by providing various interfaces to the main IDAS integrator, as well as several supporting user-callable functions. For this reason, in the following sections we refer to the *backward problem* and not to the *adjoint problem* when discussing details relevant to the DAEs that are integrated backward in time. The backward problem can be the adjoint problem (2.19) or (2.24), and can be augmented with some quadrature differential equations.

IDAS uses various constants for both input and output. These are defined as needed in this chapter, but for convenience are also listed separately in Appendix §12.

We begin with a brief overview, in the form of a skeleton user program. Following that are detailed descriptions of the interface to the various user-callable functions and of the user-supplied functions that were not already described in §5.1.

5.5.1 A skeleton of the user's main program

The following is a skeleton of the user's main program as an application of IDAS. The user program is to have these steps in the order indicated, unless otherwise noted. For the sake of brevity, we defer many of the details to the later sections. As in §5.1.3, most steps are independent of the `N_Vector`, `SUNMatrix`, `SUNLinearSolver`, and `SUNNonlinearSolver` implementations used. For the steps that are not, refer to Chapters §6, §7, §8, and §9 for the specific name of the function to be called or macro to be referenced.

Steps that are changed from the skeleton programs presented in §5.1.3, §5.4.1, and §5.4.4, are bolded.

1. Initialize parallel or multi-threaded environment
2. Create the SUNDIALS context object

Forward Problem

1. Set initial conditions for the forward problem
2. Create matrix object for the forward problem
3. Create linear solver object for the forward problem
4. Create nonlinear solver module for the forward problem
5. Create IDAS object for the forward problem
6. Initialize IDAS solver for the forward problem
7. Specify integration tolerances for forward problem
8. Attach linear solver module for the forward problem
9. Set linear solver optional inputs for the forward problem
10. Attach nonlinear solver module for the forward problem
11. Set nonlinear solver optional inputs for the forward problem
12. Initialize quadrature problem or problems for forward problems, using *IDAQuadInit()* and/or *IDAQuadSensInit()*.
13. Initialize forward sensitivity problem
14. Specify rootfinding
15. Set optional inputs for the forward problem
16. **Allocate space for the adjoint computation**

    Call *IDAAdjInit()* to allocate memory for the combined forward-backward problem. This call requires `Nd`, the number of steps between two consecutive checkpoints. *IDAAdjInit()* also specifies the type of interpolation used (see §2.6.3).

17. **Integrate forward problem**

    Call *IDASolveF()*, a wrapper for the IDAS main integration function *IDASolve()*, either in `IDA_NORMAL` mode to the time `tout` or in `IDA_ONE_STEP` mode inside a loop (if intermediate solutions of the forward problem are desired (see §5.5.2.3). The final value of `tret` is then the maximum allowable value for the endpoint T of the backward problem.

Backward Problem(s)

18. **Create vectors of endpoint values for the backward problem**

    Create the vectors `yB0` and `ypB0` at the endpoint time `tB0` $= T$ at which the backward problem starts.

19. **Create the backward problem**

    Call *IDACreateB()*, a wrapper for *IDACreate()*, to create the IDAS memory block for the new backward problem. Unlike *IDACreate()*, the function *IDACreateB()* does not return a pointer to the newly created memory block (see §5.5.2.4). Instead, this pointer is attached to the internal adjoint memory block (created by *IDAAdjInit()*) and returns an identifier called `which` that the user must later specify in any actions on the newly created backward problem.

20. **Allocate memory for the backward problem**

    Call *IDAInitB()* (or *IDAInitBS()*, when the backward problem depends on the forward sensitivities). The two functions are actually wrappers for *IDAInit()* and allocate internal memory, specify problem data, and initialize IDAS at `tB0` for the backward problem (see §5.5.2.4).

21. **Specify integration tolerances for backward problem**

    Call *IDASStolerancesB()* or *IDASVtolerancesB()* to specify a scalar relative tolerance and scalar absolute tolerance, or a scalar relative tolerance and a vector of absolute tolerances, respectively. The functions are wrappers for *IDASStolerances()* and *IDASVtolerances()* but they require an extra argument `which`, the identifier of the backward problem returned by *IDACreateB()*. See §5.5.2.5 for more information.

22. **Set optional inputs for the backward problem**

    Call `IDASet*B` functions to change from their default values any optional inputs that control the behavior of IDAS. Unlike their counterparts for the forward problem, these functions take an extra argument `which`, the identifier of the backward problem returned by *IDACreateB()* (see §5.5.2.10).

23. **Create matrix object for the backward problem**

    If a nonlinear solver requiring a linear solve will be used (e.g., the the default Newton iteration) and the linear solver will be a direct linear solver, then a template Jacobian matrix must be created by calling the appropriate constructor function defined by the particular `SUNMatrix` implementation.

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

    It is not required to use the same matrix type for both the forward and the backward problems.

    ---

24. **Create linear solver object for the backward problem**

    If a nonlinear solver requiring a linear solver is chosen (e.g., the default Newton iteration), then the desired linear solver object for the backward problem must be created by calling the appropriate constructor function defined by the particular `SUNLinearSolver` implementation.

    Note: It is not required to use the same linear solver module for both the forward and the backward problems; for example, the forward problem could be solved with the `SUNLINSOL_BAND` linear solver module and the backward problem with `SUNLINSOL_SPGMR` linear solver module.

    ---

25. **Set linear solver interface optional inputs for the backward problem**

    Call `IDASet*B` functions to change optional inputs specific to the linear solver interface. See §5.5.2.10 for details.

26. **Attach linear solver module for the backward problem**

    If a nonlinear solver requiring a linear solver is chosen for the backward problem (e.g., the default Newton iteration), then initialize the IDALS linear solver interface by attaching the linear solver object (and matrix object, if applicable) with *IDASetLinearSolverB()* (for additional details see §5.5.2.6).

27. **Create nonlinear solver object for the backward problem** (*optional*)

If using a non-default nonlinear solver for the backward problem, then create the desired nonlinear solver object by calling the appropriate constructor function defined by the particular `SUNNonlinearSolver` implementation e.g., `NLSB = SUNNonlinSol_***(...);` where `***` is the name of the nonlinear solver (see Chapter §9 for details).

28. **Attach nonlinear solver module for the backward problem** (*optional*)

    If using a non-default nonlinear solver for the backward problem, then initialize the nonlinear solver interface by attaching the nonlinear solver object by calling *IDASetNonlinearSolverB()*.

29. **Initialize quadrature calculation**

    If additional quadrature equations must be evaluated, call *IDAQuadInitB()* or *IDAQuadInitBS()* (if quadrature depends also on the forward sensitivities) as shown in §5.5.2.12. These functions are wrappers around *IDAQuadInit()* and can be used to initialize and allocate memory for quadrature integration. Optionally, call `IDASetQuad*B` functions to change from their default values optional inputs that control the integration of quadratures during the backward phase.

30. **Integrate backward problem**

    Call *IDASolveB()*, a second wrapper around the IDAS main integration function *IDASolve()*, to integrate the backward problem from `tB0`. This function can be called either in `IDA_NORMAL` or `IDA_ONE_STEP` mode. Typically, *IDASolveB()* will be called in `IDA_NORMAL` mode with an end time equal to the initial time t_0 of the forward problem.

31. **Extract quadrature variables**

    If applicable, call *IDAGetQuadB()*, a wrapper around *IDAGetQuad()*, to extract the values of the quadrature variables at the time returned by the last call to *IDASolveB()*.

32. **Deallocate memory**

    Upon completion of the backward integration, call all necessary deallocation functions. These include appropriate destructors for the vectors `y` and `yB`, a call to *IDAFree()* to free the IDAS memory block for the forward problem. If one or more additional adjoint sensitivity analyses are to be done for this problem, a call to *IDAAdjFree()* (see §5.5.2.1) may be made to free and deallocate the memory allocated for the backward problems, followed by a call to *IDAAdjInit()*.

33. Finalize MPI, if used

The above user interface to the adjoint sensitivity module in IDAS was motivated by the desire to keep it as close as possible in look and feel to the one for DAE IVP integration. Note that if steps (*18*) - (*31*) are not present, a program with the above structure will have the same functionality as one described in §5.1.3 for integration of DAEs, albeit with some overhead due to the checkpointing scheme.

If there are multiple backward problems associated with the same forward problem, repeat steps (*18*) - (*31*) above for each successive backward problem. In the process, If there are multiple backward problems associated with the same forward each call to *IDACreateB()* creates a new value of the identifier `which`.

5.5.2 User-callable functions for adjoint sensitivity analysis

5.5.2.1 Adjoint sensitivity allocation and deallocation functions

After the setup phase for the forward problem, but before the call to *IDASolveF()*, memory for the combined forward-backward problem must be allocated by a call to the function *IDAAdjInit()*. The form of the call to this function is

int **IDAAdjInit**(void *ida_mem, long int Nd, int interpType)

The function *IDAAdjInit()* updates IDAS memory block by allocating the internal memory needed for backward integration. Space is allocated for the `Nd` $= N_d$ interpolation data points, and a linked list of checkpoints is initialized.

Arguments:

- `ida_mem` – is the pointer to the IDAS memory block returned by a previous call to *IDACreate()*.
- `Nd` – is the number of integration steps between two consecutive checkpoints.
- `interpType` – specifies the type of interpolation used and can be `IDA_POLYNOMIAL` or `IDA_HERMITE`, indicating variable-degree polynomial and cubic Hermite interpolation, respectively see §2.6.3.

Return value:

- `IDA_SUCCESS` – *IDAAdjInit()* was successful.
- `IDA_MEM_FAIL` – A memory allocation request has failed.
- `IDA_MEM_NULL` – `ida_mem` was `NULL`.
- `IDA_ILL_INPUT` – One of the parameters was invalid: `Nd` was not positive or `interpType` is not one of the `IDA_POLYNOMIAL` or `IDA_HERMITE`.

Notes:

The user must set `Nd` so that all data needed for interpolation of the forward problem solution between two checkpoints fits in memory. *IDAAdjInit()* attempts to allocate space for $(2N_d + 3)$ variables of type `N_Vector`.

If an error occurred, *IDAAdjInit()* also sends a message to the error handler function.

int **IDAAdjReInit**(void *ida_mem)

The function *IDAAdjReInit()* reinitializes the IDAS memory block for ASA, assuming that the number of steps between check points and the type of interpolation remain unchanged.

Arguments:

- `ida_mem` – is the pointer to the IDAS memory block returned by a previous call to *IDACreate()*.

Return value:

- `IDA_SUCCESS` – *IDAAdjReInit()* was successful.
- `IDA_MEM_NULL` – `ida_mem` was `NULL`.
- `IDA_NO_ADJ` – The function *IDAAdjInit()* was not previously called.

Notes:

The list of check points (and associated memory) is deleted.

The list of backward problems is kept. However, new backward problems can be added to this list by calling *IDACreateB()*. If a new list of backward problems is also needed, then free the adjoint memory (by calling *IDAAdjFree()*) and reinitialize ASA with *IDAAdjInit()*.

The IDAS memory for the forward and backward problems can be reinitialized separately by calling *IDAReInit()* and *IDAReInitB()*, respectively.

void **IDAAdjFree**(void *ida_mem)

The function *IDAAdjFree()* frees the memory related to backward integration allocated by a previous call to *IDAAdjInit()*.

Arguments: The only argument is the IDAS memory block pointer returned by a previous call to *IDACreate()*.

Return value: The function *IDAAdjFree()* has no return value.

Notes:

This function frees all memory allocated by *IDAAdjInit()*. This includes workspace memory, the linked list of checkpoints, memory for the interpolation data, as well as the IDAS memory for the backward integration phase.

Unless one or more further calls to *IDAAdjInit()* are to be made, *IDAAdjFree()* should not be called by the user, as it is invoked automatically by *IDAFree()*.

5.5.2.2 Adjoint sensitivity optional input

At any time during the integration of the forward problem, the user can disable the checkpointing of the forward sensitivities by calling the following function:

int **IDAAdjSetNoSensi**(void *ida_mem)

The function *IDAAdjSetNoSensi()* instructs *IDASolveF()* not to save checkpointing data for forward sensitivities any more.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.

Return value:

- `IDA_SUCCESS` – The call to *IDACreateB()* was successful.
- `IDA_MEM_NULL` – The `ida_mem` was `NULL`.
- `IDA_NO_ADJ` – The function *IDAAdjInit()* has not been previously called.

5.5.2.3 Forward integration function

The function *IDASolveF()* is very similar to the IDAS function *IDASolve()* in that it integrates the solution of the forward problem and returns the solution $(y, \dot{y})$. At the same time, however, *IDASolveF()* stores checkpoint data every `Nd` integration steps. *IDASolveF()* can be called repeatedly by the user. Note that *IDASolveF()* is used only for the forward integration pass within an Adjoint Sensitivity Analysis. It is not for use in Forward Sensitivity Analysis; for that, see §5.4. The call to this function has the form

int **IDASolveF**(void *ida_mem, *realtype* tout, *realtype* *tret, *N_Vector* yret, *N_Vector* ypret, int itask, int *ncheck)

The function *IDASolveF()* integrates the forward problem over an interval in t and saves checkpointing data.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `tout` – the next time at which a computed solution is desired.
- `tret` – the time reached by the solver output.
- `yret` – the computed solution vector y.
- `ypret` – the computed solution vector $\dot{y}$.
- `itask` – a flag indicating the job of the solver for the next step. The `IDA_NORMAL` task is to have the solver take internal steps until it has reached or just passed the user-specified `tout` parameter. The solver then interpolates in order to return an approximate value of $y($`tout`$)$ and $\dot{y}($`tout`$)$. The `IDA_ONE_STEP` option tells the solver to take just one internal step and return the solution at the point reached by that step.
- `ncheck` – the number of internal checkpoints stored so far.

Return value:

On return, *IDASolveF()* returns vectors `yret`, `ypret` and a corresponding independent variable value `t = tret`, such that `yret` is the computed value of $y(t)$ and `ypret` the value of $\dot{y}(t)$. Additionally, it returns in `ncheck` the number of internal checkpoints saved; the total number of checkpoint intervals is `ncheck+1`. The return value flag (of type `int`) will be one of the following. For more details see the documentation for *IDASolve()*.

- `IDA_SUCCESS` – *IDASolveF()* succeeded.
- `IDA_TSTOP_RETURN` – *IDASolveF()* succeeded by reaching the optional stopping point.
- `IDA_ROOT_RETURN` – *IDASolveF()* succeeded and found one or more roots. In this case, `tret` is the location of the root. If `nrtfn` > 1 , call *IDAGetRootInfo()* to see which g_i were found to have a root.
- `IDA_NO_MALLOC` – The function *IDAInit()* has not been previously called.
- `IDA_ILL_INPUT` – One of the inputs to *IDASolveF()* is illegal.
- `IDA_TOO_MUCH_WORK` – The solver took `mxstep` internal steps but could not reach `tout`.
- `IDA_TOO_MUCH_ACC` – The solver could not satisfy the accuracy demanded by the user for some internal step.
- `IDA_ERR_FAILURE` – Error test failures occurred too many times during one internal time step or occurred with $|h| = h_{min}$.
- `IDA_CONV_FAILURE` – Convergence test failures occurred too many times during one internal time step or occurred with $|h| = h_{min}$.
- `IDA_LSETUP_FAIL` – The linear solver's setup function failed in an unrecoverable manner.
- `IDA_LSOLVE_FAIL` – The linear solver's solve function failed in an unrecoverable manner.
- `IDA_NO_ADJ` – The function *IDAAdjInit()* has not been previously called.
- `IDA_MEM_FAIL` – A memory allocation request has failed in an attempt to allocate space for a new checkpoint.

Notes:

All failure return values are negative and therefore a test `flag`< 0 will trap all *IDASolveF()* failures.

At this time, *IDASolveF()* stores checkpoint information in memory only. Future versions will provide for a safeguard option of dumping checkpoint data into a temporary file as needed. The data stored at each checkpoint is basically a snapshot of the IDAS internal memory block and contains enough information to restart the integration from that time and to proceed with the same step size and method order sequence as during the forward integration.

In addition, *IDASolveF()* also stores interpolation data between consecutive checkpoints so that, at the end of this first forward integration phase, interpolation information is already available from the last checkpoint forward. In particular, if no checkpoints were necessary, there is no need for the second forward integration phase.

Warning: It is illegal to change the integration tolerances between consecutive calls to *IDASolveF()*, as this information is not captured in the checkpoint data.

5.5.2.4 Backward problem initialization functions

The functions *IDACreateB()* and *IDAInitB()* (or *IDAInitBS()*) must be called in the order listed. They instantiate an IDAS solver object, provide problem and solution specifications, and allocate internal memory for the backward problem.

int **IDACreateB**(void *ida_mem, int *which)

The function *IDACreateB()* instantiates an IDAS solver object for the backward problem.

Arguments:

- `ida_mem` – pointer to the IDAS memory block returned by *IDACreate()*.
- `which` – contains the identifier assigned by IDAS for the newly created backward problem. Any call to `IDA*B` functions requires such an identifier.

Return value:

- `IDA_SUCCESS` – The call to *IDACreateB()* was successful.
- `IDA_MEM_NULL` – The `ida_mem` was `NULL`.
- `IDA_NO_ADJ` – The function *IDAAdjInit()* has not been previously called.
- `IDA_MEM_FAIL` – A memory allocation request has failed.

There are two initialization functions for the backward problem – one for the case when the backward problem does not depend on the forward sensitivities, and one for the case when it does. These two functions are described next.

The function *IDAInitB()* initializes the backward problem when it does not depend on the forward sensitivities. It is essentially wrapper for IDAInit with some particularization for backward integration, as described below.

int **IDAInitB**(void *ida_mem, int which, *IDAResFnB* resB, *realtype* tB0, *N_Vector* yB0, *N_Vector* ypB0)

The function *IDAInitB()* provides problem specification, allocates internal memory, and initializes the backward problem.

Arguments:

- `ida_mem` – pointer to the IDAS memory block returned by *IDACreate()*.
- `which` – represents the identifier of the backward problem.
- `resB` – is the C function which computes fB , the residual of the backward DAE problem. This function has the form `resB(t, y, yp, yB, ypB, resvalB, user_dataB)` for full details see §5.5.3.1.
- `tB0` – specifies the endpoint T where final conditions are provided for the backward problem, normally equal to the endpoint of the forward integration.
- `yB0` – is the initial value at $t =$ `tB0` of the backward solution.
- `ypB0` – is the initial derivative value at $t =$ `tB0` of the backward solution.

Return value:

- `IDA_SUCCESS` – The call to *IDAInitB()* was successful.
- `IDA_NO_MALLOC` – The function *IDAInit()* has not been previously called.
- `IDA_MEM_NULL` – The `ida_mem` was `NULL`.
- `IDA_NO_ADJ` – The function *IDAAdjInit()* has not been previously called.
- `IDA_BAD_TB0` – The final time `tB0` was outside the interval over which the forward problem was solved.

- IDA_ILL_INPUT – The parameter which represented an invalid identifier, or one of yB0 , ypB0 , resB was NULL.

Notes: The memory allocated by *IDAInitB()* is deallocated by the function *IDAAdjFree()*.

For the case when backward problem also depends on the forward sensitivities, user must call *IDAInitBS()* instead of *IDAInitB()*. Only the third argument of each function differs between these functions.

int **IDAInitBS**(void *ida_mem, int which, *IDAResFnBS* resBS, *realtype* tB0, *N_Vector* yB0, *N_Vector* ypB0)

The function *IDAInitBS()* provides problem specification, allocates internal memory, and initializes the backward problem.

Arguments:

- ida_mem – pointer to the IDAS memory block returned by *IDACreate()*.
- which – represents the identifier of the backward problem.
- resBS – is the C function which computes fB , the residual or the backward DAE problem. This function has the form resBS(t, y, yp, yS, ypS, yB, ypB, resvalB, user_dataB) for full details see §5.5.3.2.
- tB0 – specifies the endpoint T where final conditions are provided for the backward problem.
- yB0 – is the initial value at $t =$ tB0 of the backward solution.
- ypB0 – is the initial derivative value at $t =$ tB0 of the backward solution.

Return value:

- IDA_SUCCESS – The call to *IDAInitB()* was successful.
- IDA_NO_MALLOC – The function *IDAInit()* has not been previously called.
- IDA_MEM_NULL – The ida_mem was NULL.
- IDA_NO_ADJ – The function *IDAAdjInit()* has not been previously called.
- IDA_BAD_TB0 – The final time tB0 was outside the interval over which the forward problem was solved.
- IDA_ILL_INPUT – The parameter which represented an invalid identifier, or one of yB0 , ypB0 , resB was NULL , or sensitivities were not active during the forward integration.

Notes: The memory allocated by *IDAInitBS()* is deallocated by the function *IDAAdjFree()*.

The function *IDAReInitB()* reinitializes idas for the solution of a series of backward problems, each identified by a value of the parameter which. *IDAReInitB()* is essentially a wrapper for *IDAReInit()*, and so all details given for *IDAReInit()* apply here. Also, *IDAReInitB()* can be called to reinitialize a backward problem even if it has been initialized with the sensitivity-dependent version *IDAInitBS()*. Before calling *IDAReInitB()* for a new backward problem, call any desired solution extraction functions IDAGet** associated with the previous backward problem. The call to the *IDAReInitB()* function has the form

int **IDAReInitB**(void *ida_mem, int which, *realtype* tB0, *N_Vector* yB0, *N_Vector* ypB0)

The function *IDAReInitB()* reinitializes an IDAS backward problem.

Arguments:

- ida_mem – pointer to IDAS memory block returned by *IDACreate()*.
- which – represents the identifier of the backward problem.
- tB0 – specifies the endpoint T where final conditions are provided for the backward problem.
- yB0 – is the initial value at $t =$ tB0 of the backward solution.
- ypB0 – is the initial derivative value at $t =$ tB0 of the backward solution.

Return value:

- IDA_SUCCESS – The call to *IDAReInitB()* was successful.
- IDA_NO_MALLOC – The function *IDAInit()* has not been previously called.
- IDA_MEM_NULL – The ida_mem memory block pointer was NULL.
- IDA_NO_ADJ – The function *IDAAdjInit()* has not been previously called.
- IDA_BAD_TB0 – The final time tB0 is outside the interval over which the forward problem was solved.
- IDA_ILL_INPUT – The parameter which represented an invalid identifier, or one of yB0 , ypB0 was NULL.

5.5.2.5 Tolerance specification functions for backward problem

One of the following two functions must be called to specify the integration tolerances for the backward problem. Note that this call must be made after the call to *IDAInitB()* or *IDAInitBS()*.

int **IDASStolerancesB**(void *ida_mem, int which, *realtype* reltolB, *realtype* abstolB)
The function *IDASStolerancesB()* specifies scalar relative and absolute tolerances.

Arguments:

- ida_mem – pointer to the IDAS memory block returned by *IDACreate()*.
- which – represents the identifier of the backward problem.
- reltolB – is the scalar relative error tolerance.
- abstolB – is the scalar absolute error tolerance.

Return value:

- IDA_SUCCESS – The call to *IDASStolerancesB()* was successful.
- IDA_MEM_NULL – The IDAS memory block was not initialized through a previous call to *IDACreate()*.
- IDA_NO_MALLOC – The allocation function *IDAInit()* has not been called.
- IDA_NO_ADJ – The function *IDAAdjInit()* has not been previously called.
- IDA_ILL_INPUT – One of the input tolerances was negative.

int **IDASVtolerancesB**(void *ida_mem, int which, *realtype* reltolB, *N_Vector* abstolB)
The function *IDASVtolerancesB()* specifies scalar relative tolerance and vector absolute tolerances.

Arguments:

- ida_mem – pointer to the IDAS memory block returned by *IDACreate()*.
- which – represents the identifier of the backward problem.
- reltolB – is the scalar relative error tolerance.
- abstolB – is the vector of absolute error tolerances.

Return value:

- IDA_SUCCESS – The call to *IDASVtolerancesB()* was successful.
- IDA_MEM_NULL – The IDAS memory block was not initialized through a previous call to *IDACreate()*.
- IDA_NO_MALLOC – The allocation function *IDAInit()* has not been called.

- IDA_NO_ADJ – The function *IDAAdjInit()* has not been previously called.
- IDA_ILL_INPUT – The relative error tolerance was negative or the absolute tolerance had a negative component.

Notes: This choice of tolerances is important when the absolute error tolerance needs to be different for each component of the DAE state vector y.

5.5.2.6 Linear solver initialization functions for backward problem

All IDAS linear solver modules available for forward problems are available for the backward problem. They should be created as for the forward problem then attached to the memory structure for the backward problem using the following function.

int **IDASetLinearSolverB**(void *ida_mem, int which, *SUNLinearSolver* LS, *SUNMatrix* A)

The function *IDASetLinearSolverB()* attaches a generic SUNLinearSolver object LS and corresponding template Jacobian SUNMatrix object A (if applicable) to IDAS, initializing the IDALS linear solver interface for solution of the backward problem.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- which – represents the identifier of the backward problem returned by *IDACreateB()*.
- LS – SUNLinearSolver object to use for solving linear systems for the backward problem.
- A – SUNMatrix object for used as a template for the Jacobian for the backward problem or NULL if not applicable.

Return value:

- IDALS_SUCCESS – The IDALS initialization was successful.
- IDALS_MEM_NULL – The ida_mem pointer is NULL.
- IDALS_ILL_INPUT – The parameter which represented an invalid identifier.
- IDALS_MEM_FAIL – A memory allocation request failed.
- IDALS_NO_ADJ – The function *IDAAdjInit()* has not been previously called.

Notes:

If LS is a matrix-based linear solver, then the template Jacobian matrix A 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 IDADlsSetLinearSolverB and IDASpilsSetLinearSolverB are now deprecated.

5.5.2.7 Nonlinear solver initialization functions for backward problem

As with the forward problem IDAS uses the SUNNonlinearSolver implementation of Newton's method defined by the SUNNONLINSOL_NEWTON module (see §9.3) by default.

To specify a different nonlinear solver in IDAS for the backward problem, the user's program must create a SUNNonlinearSolver object by calling the appropriate constructor routine. The user must then attach the SUNNonlinearSolver object to IDAS by calling *IDASetNonlinearSolverB()*, as documented below.

When changing the nonlinear solver in IDAS, *IDASetNonlinearSolverB()* must be called after *IDAInitB()*. If any calls to *IDASolveB()* have been made, then IDAS will need to be reinitialized by calling *IDAReInitB()* to ensure that the nonlinear solver is initialized correctly before any subsequent calls to *IDASolveB()*.

int **IDASetNonlinearSolverB**(void *ida_mem, int which, *SUNNonlinearSolver* NLS)

The function `IDASetNonLinearSolverB()` attaches a `SUNNonlinearSolver` object (`NLS`) to IDAS for the solution of the backward problem.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `which` – represents the identifier of the backward problem returned by *IDACreateB()*.
- `NLS` – SUNNonlinearSolver object to use for solving nonlinear systems for the backward problem.

Return value:

- `IDA_SUCCESS` – The nonlinear solver was successfully attached.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDALS_NO_ADJ` – The function `IDAAdjInit` has not been previously called.
- `IDA_ILL_INPUT` – The parameter `which` represented an invalid identifier or the SUNNonlinearSolver object is `NULL` , does not implement the required nonlinear solver operations, is not of the correct type, or the residual function, convergence test function, or maximum number of nonlinear iterations could not be set.

5.5.2.8 Initial condition calculation functions for backward problem

IDAS provides support for calculation of consistent initial conditions for certain backward index-one problems of semi-implicit form through the functions *IDACalcICB()* and *IDACalcICBS()*. Calling them is optional. It is only necessary when the initial conditions do not satisfy the adjoint system.

The above functions provide the same functionality for backward problems as *IDACalcIC()* with parameter `icopt = IDA_YA_YDP_INIT` provides for forward problems: compute the algebraic components of yB and differential components of $\dot{y}B$, given the differential components of yB. They require that the `IDASetIdB()` was previously called to specify the differential and algebraic components.

Both functions require forward solutions at the final time `tB0`. *IDACalcICBS()* also needs forward sensitivities at the final time `tB0`.

int **IDACalcICB**(void *ida_mem, int which, *realtype* tBout1, *N_Vector* yfin, *N_Vector* ypfin)

The function *IDACalcICB()* corrects the initial values `yB0` and `ypB0` at time `tB0` for the backward problem.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `which` – is the identifier of the backward problem.
- `tBout1` – is the first value of t at which a solution will be requested from *IDASolveB()*. This value is needed here only to determine the direction of integration and rough scale in the independent variable t.
- `yfin` – the forward solution at the final time `tB0`.
- `ypfin` – the forward solution derivative at the final time `tB0`.

Return value:

- `IDA_NO_ADJ` – *IDAAdjInit()* has not been previously called.

- IDA_ILL_INPUT – Parameter which represented an invalid identifier.

Notes:

All failure return values are negative and therefore a test flag < 0 will trap all *IDACalcICB()* failures. Note that *IDACalcICB()* will correct the values of $yB(tB_0)$ and $\dot{y}B(tB_0)$ which were specified in the previous call to *IDAInitB()* or *IDAReInitB()*. To obtain the corrected values, call IDAGetconsistentICB() (see §5.5.2.11).

IDACalcICB() will correct the values of $yB(tB_0)$ and $\dot{y}B(tB_0)$ which were specified in the previous call to *IDAInitB()* or *IDAReInitB()*. To obtain the corrected values, :call c:func:*IDAGetConsistentICB* (see :§5.5.2.11).

In the case where the backward problem also depends on the forward sensitivities, user must call the following function to correct the initial conditions:

int **IDACalcICBS**(void *ida_mem, int which, *realtype* tBout1, *N_Vector* yfin, *N_Vector* ypfin, *N_Vector* ySfin, *N_Vector* ypSfin)

The function *IDACalcICBS()* corrects the initial values yB0 and ypB0 at time tB0 for the backward problem.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- which – is the identifier of the backward problem.
- tBout1 – is the first value of t at which a solution will be requested from *IDASolveB()* .This value is needed here only to determine the direction of integration and rough scale in the independent variable t.
- yfin – the forward solution at the final time tB0.
- ypfin – the forward solution derivative at the final time tB0.
- ySfin – a pointer to an array of Ns vectors containing the sensitivities of the forward solution at the final time tB0.
- ypSfin – a pointer to an array of Ns vectors containing the derivatives of the forward solution sensitivities at the final time tB0.

Return value:

- IDA_NO_ADJ – *IDAAdjInit()* has not been previously called.
- IDA_ILL_INPUT – Parameter which represented an invalid identifier, sensitivities were not active during forward integration, or *IDAInitBS()* or IDAReInitBS() has not been previously called.

Notes:

All failure return values are negative and therefore a test flag < 0 will trap all *IDACalcICBS()* failures. Note that *IDACalcICBS()* will correct the values of $yB(tB_0)$ and $\dot{y}B(tB_0)$ which were specified in the previous call to *IDAInitBS()* or IDAReInitBS(). To obtain the corrected values, call *IDAGetConsistentICB()* (see §5.5.2.11).

IDACalcICBS() will correct the values of $yB(tB_0)$ and $\dot{y}B(tB_0)$ which were specified in the previous call to *IDAInitBS()* or IDAReInitBS(). To obtain the corrected values, :call *IDAGetConsistentICB()*.

5.5.2.9 Backward integration function

The function *IDASolveB()* performs the integration of the backward problem. It is essentially a wrapper for the IDAS main integration function *IDASolve()* and, in the case in which checkpoints were needed, it evolves the solution of the backward problem through a sequence of forward-backward integration pairs between consecutive checkpoints. In each pair, the first run integrates the original IVP forward in time and stores interpolation data; the second run integrates the backward problem backward in time and performs the required interpolation to provide the solution of the IVP to the backward problem.

The function *IDASolveB()* does not return the solution `yB` itself. To obtain that, call the function *IDAGetB()*, which is also described below.

The *IDASolveB()* function does not support rootfinding, unlike `IDASoveF()`, which supports the finding of roots of functions of $(t, y, \dot{y})$. If rootfinding was performed by *IDASolveF()*, then for the sake of efficiency, it should be disabled for *IDASolveB()* by first calling *IDARootInit()* with `nrtfn` = 0.

The call to *IDASolveB()* has the form

int **IDASolveB**(void *ida_mem, *realtype* tBout, int itaskB)

The function *IDASolveB()* integrates the backward DAE problem.

Arguments:

- `ida_mem` – pointer to the IDAS memory returned by *IDACreate()*.
- `tBout` – the next time at which a computed solution is desired.
- `itaskB` – output mode a flag indicating the job of the solver for the next step. The `IDA_NORMAL` task is to have the solver take internal steps until it has reached or just passed the user-specified value `tBout`. The solver then interpolates in order to return an approximate value of yB(`tBout`). The `IDA_ONE_STEP` option tells the solver to take just one internal step in the direction of `tBout` and return.

Return value:

- `IDA_SUCCESS` – *IDASolveB()* succeeded.
- `IDA_MEM_NULL` – The `ida_mem` was `NULL`.
- `IDA_NO_ADJ` – The function *IDAAdjInit()* has not been previously called.
- `IDA_NO_BCK` – No backward problem has been added to the list of backward problems by a call to *IDACreateB()*.
- `IDA_NO_FWD` – The function *IDASolveF()* has not been previously called.
- `IDA_ILL_INPUT` – One of the inputs to *IDASolveB()* is illegal.
- `IDA_BAD_ITASK` – The `itaskB` argument has an illegal value.
- `IDA_TOO_MUCH_WORK` – The solver took `mxstep` internal steps but could not reach `tBout`.
- `IDA_TOO_MUCH_ACC` – The solver could not satisfy the accuracy demanded by the user for some internal step.
- `IDA_ERR_FAILURE` – Error test failures occurred too many times during one internal time step.
- `IDA_CONV_FAILURE` – Convergence test failures occurred too many times during one internal time step.
- `IDA_LSETUP_FAIL` – The linear solver's setup function failed in an unrecoverable manner.
- `IDA_SOLVE_FAIL` – The linear solver's solve function failed in an unrecoverable manner.
- `IDA_BCKMEM_NULL` – The IDAS memory for the backward problem was not created with a call to *IDACreateB()*.

- IDA_BAD_TBOUT – The desired output time tBout is outside the interval over which the forward problem was solved.
- IDA_REIFWD_FAIL – Reinitialization of the forward problem failed at the first checkpoint corresponding to the initial time of the forward problem.
- IDA_FWD_FAIL – An error occurred during the integration of the forward problem.

Notes: All failure return values are negative and therefore a test flag< 0 will trap all *IDASolveB()* failures. In the case of multiple checkpoints and multiple backward problems, a given call to *IDASolveB()* in IDA_ONE_STEP mode may not advance every problem one step, depending on the relative locations of the current times reached. But repeated calls will eventually advance all problems to tBout.

To obtain the solution yB to the backward problem, call the function *IDAGetB()* as follows:

int **IDAGetB**(void *ida_mem, int which, *realtype* *tret, *N_Vector* yB, *N_Vector* ypB)
The function *IDAGetB()* provides the solution yB of the backward DAE problem.

Arguments:

- ida_mem – pointer to the IDAS memory returned by *IDACreate()*.
- which – the identifier of the backward problem.
- tret – the time reached by the solver output.
- yB – the backward solution at time tret.
- ypB – the backward solution derivative at time tret.

Return value:

- IDA_SUCCESS – *IDAGetB()* was successful.
- IDA_MEM_NULL – ida_mem is NULL.
- IDA_NO_ADJ – The function *IDAAdjInit()* has not been previously called.
- IDA_ILL_INPUT – The parameter which is an invalid identifier.

Notes: To obtain the solution associated with a given backward problem at some other time within the last integration step, first obtain a pointer to the proper IDAS memory structure by calling *IDAGetAdjIDABmem()* and then use it to call *IDAGetDky()*.

Warning: The user must allocate space for yB and ypB.

5.5.2.10 Optional input functions for the backward problem

As for the forward problem there are numerous optional input parameters that control the behavior of the IDAS solver for the backward problem. IDAS provides functions that can be used to change these optional input parameters from their default values which are then described in detail in the remainder of this section, beginning with those for the main IDAS solver and continuing with those for the linear solver interfaces. For the most casual use of IDAS, the reader can skip to §5.5.3.

We note that, on an error return, all of the optional input functions send an error message to the error handler function. All error return values are negative, so the test flag < 0 will catch all errors. Finally, a call to a IDASet***B function can be made from the user's calling program at any time and, if successful, takes effect immediately.

Main solver optional input functions

The adjoint module in IDAS provides wrappers for most of the optional input functions defined in §5.1.4.10. The only difference is that the user must specify the identifier `which` of the backward problem within the list managed by IDAS.

The optional input functions defined for the backward problem are:

```
flag = IDASetUserDataB(ida_mem, which, user_dataB);
flag = IDASetMaxOrdB(ida_mem, which, maxordB);
flag = IDASetMaxNumStepsB(ida_mem, which, mxstepsB);
flag = IDASetInitStepB(ida_mem, which, hinB)
flag = IDASetMaxStepB(ida_mem, which, hmaxB);
flag = IDASetSuppressAlgB(ida_mem, which, suppressalgB);
flag = IDASetIdB(ida_mem, which, idB);
flag = IDASetConstraintsB(ida_mem, which, constraintsB);
```

Their return value `flag` (of type `int`) can have any of the return values of their counterparts, but it can also be `IDA_NO_ADJ` if *IDAAdjInit()* has not been called, or `IDA_ILL_INPUT` if `which` was an invalid identifier.

Linear solver interface optional input functions

When using matrix-based linear solver modules for the backward problem, i.e., a non-`NULL` `SUNMatrix` object `A` was passed to *IDASetLinearSolverB()*, the IDALS linear solver interface needs a function to compute an approximation to the Jacobian matrix. This can be attached through a call to either *IDASetJacFnB()* or *IDASetJacFnBS()*, with the second used when the backward problem depends on the forward sensitivities.

int **IDASetJacFnB**(void *ida_mem, int which, *IDALsJacFnB* jacB)

The function *IDASetJacFnB()* specifies the Jacobian approximation function to be used for the backward problem.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `which` – represents the identifier of the backward problem.
- `jacB` – user-defined Jacobian approximation function.

Return value:

- `IDALS_SUCCESS` – *IDASetJacFnB()* succeeded.
- `IDALS_MEM_NULL` – The `ida_mem` was `NULL`.
- `IDALS_NO_ADJ` – The function *IDAAdjInit()* has not been previously called.
- `IDALS_LMEM_NULL` – The linear solver has not been initialized with a call to *IDASetLinearSolverB()*.
- `IDALS_ILL_INPUT` – The parameter `which` represented an invalid identifier.

Notes: The previous routine `IDADlsSetJacFnB` 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.

int **IDASetJacFnBS**(void *ida_mem, int which, *IDALsJacFnBS* jacBS)

The function *IDASetJacFnBS()* specifies the Jacobian approximation function to be used for the backward problem in the case where the backward problem depends on the forward sensitivities.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- which – represents the identifier of the backward problem.
- jacBS – user-defined Jacobian approximation function.

Return value:

- IDALS_SUCCESS – *IDASetJacFnBS()* succeeded.
- IDALS_MEM_NULL – The ida_mem was NULL.
- IDALS_NO_ADJ – The function *IDAAdjInit()* has not been previously called.
- IDALS_LMEM_NULL – The linear solver has not been initialized with a call to IDASetLinearSolverBS().
- IDALS_ILL_INPUT – The parameter which represented an invalid identifier.

Notes: The previous routine, IDADlsSetJacFnBS, is now deprecated.

The function *IDASetLinearSolutionScalingB()* can be used to enable or disable solution scaling when using a matrix-based linear solver.

int **IDASetLinearSolutionScalingB**(void *ida_mem, int which, *booleantype* onoffB)

The function *IDASetLinearSolutionScalingB()* enables or disables scaling the linear system solution to account for a change in α in the linear system in the backward problem. For more details see §8.2.1.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- which – represents the identifier of the backward problem.
- onoffB – flag to enable SUNTRUE or disable SUNFALSE scaling.

Return value:

- IDALS_SUCCESS – The flag value has been successfully set.
- IDALS_MEM_NULL – The ida_mem pointer is NULL.
- IDALS_LMEM_NULL – The IDALS linear solver interface has not been initialized.
- IDALS_ILL_INPUT – The attached linear solver is not matrix-based.

Notes:

By default scaling is enabled with matrix-based linear solvers when using BDF methods.

By default scaling is enabled with matrix-based linear solvers when using BDF methods.

When using a matrix-free linear solver module for the backward problem, the IDALS linear solver interface requires a function to compute an approximation to the product between the Jacobian matrix $J(t, y)$ and a vector v. This may be performed internally using a difference-quotient approximation, or it may be supplied by the user by calling one of the following two functions:

int **IDASetJacTimesB**(void *ida_mem, int which, *IDALsJacTimesSetupFnB* jsetupB, *IDALsJacTimesVecFnB* jtimesB)

The function *IDASetJacTimesB()* specifies the Jacobian-vector setup and product functions to be used.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- which – the identifier of the backward problem.

- jtsetupB – user-defined function to set up the Jacobian-vector product. Pass NULL if no setup is necessary.
- jtimesB – user-defined Jacobian-vector product function.

Return value:

- IDALS_SUCCESS – The optional value has been successfully set.
- IDALS_MEM_NULL – The ida_mem memory block pointer was NULL.
- IDALS_LMEM_NULL – The IDALS linear solver has not been initialized.
- IDALS_NO_ADJ – The function *IDAAdjInit()* has not been previously called.
- IDALS_ILL_INPUT – The parameter which represented an invalid identifier.

Warning: The previous routine, IDASpilsSetJacTimesB, is now deprecated.

int **IDASetJacTimesBS**(void *ida_mem, int which, *IDALsJacTimesSetupFnBS* jsetupBS, *IDALsJacTimesVecFnBS* jtimesBS)

The function *IDASetJacTimesBS()* specifies the Jacobian-vector product setup and evaluation functions to be used, in the case where the backward problem depends on the forward sensitivities.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- which – the identifier of the backward problem.
- jtsetupBS – user-defined function to set up the Jacobian-vector product. Pass NULL if no setup is necessary.
- jtimesBS – user-defined Jacobian-vector product function.

Return value:

- IDALS_SUCCESS – The optional value has been successfully set.
- IDALS_MEM_NULL – The ida_mem memory block pointer was NULL.
- IDALS_LMEM_NULL – The IDALS linear solver has not been initialized.
- IDALS_NO_ADJ – The function *IDAAdjInit()* has not been previously called.
- IDALS_ILL_INPUT – The parameter which represented an invalid identifier.

Warning: The previous routine, IDASpilsSetJacTimesBS, is now deprecated.

When using the default difference-quotient approximation to the Jacobian-vector product for the backward problem, the user may specify the factor to use in setting increments for the finite-difference approximation, via a call to *IDASetIncrementFactorB()*.

int **IDASetIncrementFactorB**(void *ida_mem, int which, *realtype* dqincfacB)

The function *IDASetIncrementFactorB()* specifies the factor in the increments used in the difference quotient approximations to matrix-vector products for the backward problem. This routine can be used in both the cases where the backward problem does and does not depend on the forward sensitvities.

Arguments:

- ida_mem – pointer to the IDAS memory block.

- `which` – the identifier of the backward problem.
- `dqincfacB` – difference quotient approximation factor.

Return value:

- `IDALS_SUCCESS` – The optional value has been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.
- `IDALS_NO_ADJ` – The function *IDAAdjInit()* has not been previously called.
- `IDALS_ILL_INPUT` – The parameter `which` represented an invalid identifier.

Notes: The default value is 1.0.

The previous routine `IDASpilsSetIncrementFactorB` is now a deprecated.

Additionally, When using the internal difference quotient for the backward problem, the user may also optionally supply an alternative residual function for use in the Jacobian-vector product approximation by calling *IDASetJacTimesResFnB()*. The alternative residual side function should compute a suitable (and differentiable) approximation to the residual function provided to *IDAInitB()* or *IDAInitBS()*. For example, as done in [22] for the forward integration of an ODE in explicit form without sensitivity analysis, the alternative function may use lagged values when evaluating a nonlinearity in the right-hand side to avoid differencing a potentially non-differentiable factor.

int **IDASetJacTimesResFnB**(void *ida_mem, int which, *IDAResFn* jtimesResFn)

The function *IDASetJacTimesResFnB()* specifies an alternative DAE residual function for use in the internal Jacobian-vector product difference quotient approximation for the backward problem.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `which` – the identifier of the backward problem.
- `jtimesResFn` – is the C function which computes the alternative DAE residual function to use in Jacobian-vector product difference quotient approximations. This function has the form `res(t, yy, yp, resval, user_data)`. For full details see §5.1.5.1.

Return value:

- `IDALS_SUCCESS` – The optional value has been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.
- `IDALS_NO_ADJ` – The function *IDAAdjInit()* has not been previously called.
- `IDALS_ILL_INPUT` – The parameter `which` represented an invalid identifier or the internal difference quotient approximation is disabled.

Notes: The default is to use the residual function provided to *IDAInit()* in the internal difference quotient. If the input resudual function is `NULL`, the default is used.

This function must be called *after* the IDALS linear solver interface has been initialized through a call to *IDASetLinearSolverB()*.

When using an iterative linear solver for the backward problem, the user may supply a preconditioning operator to aid in solution of the system, or she/he may adjust the convergence tolerance factor for the iterative linear solver. These may be accomplished through calling the following functions:

int **IDASetPreconditionerB**(void *ida_mem, int which, *IDALsPrecSetupFnB* psetupB, *IDALsPrecSolveFnB* psolveB)

The function `IDASetPrecSolveFnB()` specifies the preconditioner setup and solve functions for the backward integration.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `which` – the identifier of the backward problem.
- `psetupB` – user-defined preconditioner setup function.
- `psolveB` – user-defined preconditioner solve function.

Return value:

- `IDALS_SUCCESS` – The optional value has been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` memory block pointer was `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.
- `IDALS_NO_ADJ` – The function *IDAAdjInit()* has not been previously called.
- `IDALS_ILL_INPUT` – The parameter `which` represented an invalid identifier.

Notes: The `psetupB` argument may be `NULL` if no setup operation is involved in the preconditioner.

Warning: The previous routine `IDASpilsSetPreconditionerB` is now deprecated.

int **IDASetPreconditionerBS**(void *ida_mem, int which, *IDALsPrecSetupFnBS* psetupBS, *IDALsPrecSolveFnBS* psolveBS)

The function `IDASetPrecSolveFnBS()` specifies the preconditioner setup and solve functions for the backward integration, in the case where the backward problem depends on the forward sensitivities.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `which` – the identifier of the backward problem.
- `psetupBS` – user-defined preconditioner setup function.
- `psolveBS` – user-defined preconditioner solve function.

Return value:

- `IDALS_SUCCESS` – The optional value has been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` memory block pointer was `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.
- `IDALS_NO_ADJ` – The function *IDAAdjInit()* has not been previously called.
- `IDALS_ILL_INPUT` – The parameter `which` represented an invalid identifier.

Notes: The `psetupBS` argument may be `NULL` if no setup operation is involved in the preconditioner.

Warning: The previous routine `IDASpilsSetPreconditionerBS` is now deprecated.

int **IDASetEpsLinB**(void *ida_mem, int which, *realtype* eplifacB)

The function *IDASetEpsLinB()* specifies the factor by which the Krylov linear solver's convergence test constant is reduced from the nonlinear iteration test constant. (See §2.1). This routine can be used in both the cases wherethe backward problem does and does not depend on the forward sensitvities.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `which` – the identifier of the backward problem.
- `eplifacB` – linear convergence safety factor $>= 0.0$.

Return value:

- `IDALS_SUCCESS` – The optional value has been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.
- `IDALS_NO_ADJ` – The function *IDAAdjInit()* has not been previously called.
- `IDALS_ILL_INPUT` – The parameter `which` represented an invalid identifier.

Notes: The default value is 0.05.

Passing a value `eplifacB` $= 0.0$ also indicates using the default value.

Warning: The previous routine `IDASpilsSetEpsLinB` is now deprecated.

int **IDASetLSNormFactorB**(void *ida_mem, int which, *realtype* nrmfac)

The function *IDASetLSNormFactorB()* specifies the factor to use when converting from the integrator tolerance (WRMS norm) to the linear solver tolerance (L2 norm) for Newton linear system solves e.g., `tol_L2 = fac * tol_WRMS`. This routine can be used in both the cases wherethe backward problem does and does not depend on the forward sensitvities.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `which` – the identifier of the backward problem.
- `nrmfac` – the norm conversion factor. If `nrmfac` is:
  - > 0 then the provided value is used.
  - $= 0$ then the conversion factor is computed using the vector length i.e., `nrmfac = N_VGetLength(y)` default.
  - < 0 then the conversion factor is computed using the vector dot product `nrmfac = N_VDotProd(v,v)` where all the entries of `v` are one.

Return value:

- `IDALS_SUCCESS` – The optional value has been successfully set.
- `IDALS_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDALS_LMEM_NULL` – The IDALS linear solver has not been initialized.
- `IDALS_NO_ADJ` – The function *IDAAdjInit()* has not been previously called.
- `IDALS_ILL_INPUT` – The parameter `which` represented an invalid identifier.

Notes: This function must be called after the IDALS linear solver interface has been initialized through a call to *IDASetLinearSolverB()*.

Prior to the introduction of `N_VGetLength` in SUNDIALS v5.0.0 (IDAS v4.0.0) the value of `nrmfac` was computed using the vector dot product i.e., the `nrmfac < 0` case.

5.5.2.11 Optional output functions for the backward problem

Main solver optional output functions

The user of the adjoint module in IDAS has access to any of the optional output functions described in §5.1.4.12, both for the main solver and for the linear solver modules. The first argument of these `IDAGet*` and `IDA*Get*` functions is the pointer to the IDAS memory block for the backward problem. In order to call any of these functions, the user must first call the following function to obtain this pointer:

int **IDAGetAdjIDABmem**(void *ida_mem, int which)

The function *IDAGetAdjIDABmem()* returns a pointer to the IDAS memory block for the backward problem.

Arguments:

- `ida_mem` – pointer to the IDAS memory block created by *IDACreate()*.
- `which` – the identifier of the backward problem.

Return value:

- The return value, `ida_memB` (of type `void *`), is a pointer to the idas memory for the backward problem.

Warning: The user should not modify `ida_memB` in any way.

Optional output calls should pass `ida_memB` as the first argument; thus, for example, to get the number of integration steps: `flag = IDAGetNumSteps(idas_memB,&nsteps)`.

To get values of the *forward* solution during a backward integration, use the following function. The input value of `t` would typically be equal to that at which the backward solution has just been obtained with *IDAGetB()*. In any case, it must be within the last checkpoint interval used by *IDASolveB()*.

int **IDAGetAdjY**(void *ida_mem, *realtype* t, *N_Vector* y, *N_Vector* yp)

The function *IDAGetAdjY()* returns the interpolated value of the forward solution y and its derivative during a backward integration.

Arguments:

- `ida_mem` – pointer to the IDAS memory block created by *IDACreate()*.
- `t` – value of the independent variable at which y is desired input.
- `y` – forward solution $y(t)$.
- `yp` – forward solution derivative $\dot{y}(t)$.

Return value:

- `IDA_SUCCESS` – *IDAGetAdjY()* was successful.
- `IDA_MEM_NULL` – `ida_mem` was `NULL`.
- `IDA_GETY_BADT` – The value of `t` was outside the current checkpoint interval.

Warning: The user must allocate space for `y` and `yp`.

int **IDAGetAdjCheckPointsInfo**(void *ida_mem, IDAadjCheckPointRec *ckpnt)

The function *IDAGetAdjCheckPointsInfo()* loads an array of `ncheck+1` records of type `IDAadjCheckPointRec()`. The user must allocate space for the array `ckpnt`.

Arguments:

- `ida_mem` – pointer to the IDAS memory block created by *IDACreate()*.
- `ckpnt` – array of `ncheck+1` checkpoint records, each of type `IDAadjCheckPointRec()`.

Return value:

- `void`

Notes: The members of each record `ckpnt[i]` are:

- `ckpnt[i].my_addr` (`void *`) address of current checkpoint in `ida_mem->ida_adj_mem`
- `ckpnt[i].next_addr` (`void *`) address of next checkpoint
- `ckpnt[i].t0` (`realtype`) start of checkpoint interval
- `ckpnt[i].t1` (`realtype`) end of checkpoint interval
- `ckpnt[i].nstep` (`long int`) step counter at ckeckpoint `t0`
- `ckpnt[i].order` (`int`) method order at checkpoint `t0`
- `ckpnt[i].step` (`realtype`) step size at checkpoint `t0`

Initial condition calculation optional output function

int **IDAGetConsistentICB**(void *ida_mem, int which, *N_Vector* yB0_mod, *N_Vector* ypB0_mod)

The function *IDAGetConsistentICB()* returns the corrected initial conditions for backward problem calculated by *IDACalcICB()*.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `which` – is the identifier of the backward problem.
- `yB0_mod` – consistent initial vector.
- `ypB0_mod` – consistent initial derivative vector.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.
- `IDA_NO_ADJ` – *IDAAdjInit()* has not been previously called.
- `IDA_ILL_INPUT` – Parameter `which` did not refer a valid backward problem identifier.

Notes: If the consistent solution vector or consistent derivative vector is not desired, pass `NULL` for the corresponding argument.

Warning: The user must allocate space for `yB0_mod` and `ypB0_mod` (if not `NULL`).

5.5.2.12 Backward integration of quadrature equations

Not only the backward problem but also the backward quadrature equations may or may not depend on the forward sensitivities. Accordingly, one of the *IDAQuadInitB()* or *IDAQuadInitBS()* should be used to allocate internal memory and to initialize backward quadratures. For any other operation (extraction, optional input/output, reinitialization, deallocation), the same function is called regardless of whether or not the quadratures are sensitivity-dependent.

Backward quadrature initialization functions

The function *IDAQuadInitB()* initializes and allocates memory for the backward integration of quadrature equations that do not depende on forward sensititvities. It has the following form:

int **IDAQuadInitB**(void *ida_mem, int which, *IDAQuadRhsFnB* rhsQB, *N_Vector* yQB0)

The function *IDAQuadInitB()* provides required problem specifications, allocates internal memory, and initializes backward quadrature integration.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `which` – the identifier of the backward problem.
- `rhsQB` – is the C function which computes fQB , the residual of the backward quadrature equations. This function has the form `rhsQB(t, y, yp, yB, ypB, rhsvalBQ, user_dataB)` see §5.5.3.3.
- `yQB0` – is the value of the quadrature variables at `tB0`.

Return value:

- `IDA_SUCCESS` – The call to *IDAQuadInitB()* was successful.
- `IDA_MEM_NULL` – `ida_mem` was `NULL`.
- `IDA_NO_ADJ` – The function *IDAAdjInit()* has not been previously called.
- `IDA_MEM_FAIL` – A memory allocation request has failed.
- `IDA_ILL_INPUT` – The parameter `which` is an invalid identifier.

int **IDAQuadInitBS**(void *ida_mem, int which, *IDAQuadRhsFnBS* rhsQBS, *N_Vector* yQBS0)

The function *IDAQuadInitBS()* provides required problem specifications, allocates internal memory, and initializes backward quadrature integration with sensitivities.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `which` – the identifier of the backward problem.
- `rhsQBS` – is the C function which computes $fQBS$, the residual of the backward quadrature equations. This function has the form `rhsQBS(t, y, yp, yS, ypS, yB, ypB, rhsvalBQS, user_dataB)` see §5.5.3.4.
- `yQBS0` – is the value of the sensitivity-dependent quadrature variables at `tB0`.

Return value:

- `IDA_SUCCESS` – The call to *IDAQuadInitBS()* was successful.

- IDA_MEM_NULL – ida_mem was NULL.
- IDA_NO_ADJ – The function *IDAAdjInit()* has not been previously called.
- IDA_MEM_FAIL – A memory allocation request has failed.
- IDA_ILL_INPUT – The parameter which is an invalid identifier.

The integration of quadrature equations during the backward phase can be re-initialized by calling the following function. Before calling *IDAQuadReInitB()* for a new backward problem, call any desired solution extraction functions IDAGet** associated with the previous backward problem.

int **IDAQuadReInitB**(void *ida_mem, int which, *N_Vector* yQB0)

The function *IDAQuadReInitB()* re-initializes the backward quadrature integration.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- which – the identifier of the backward problem.
- yQB0 – is the value of the quadrature variables at tB0.

Return value:

- IDA_SUCCESS – The call to *IDAQuadReInitB()* was successful.
- IDA_MEM_NULL – ida_mem was NULL.
- IDA_NO_ADJ – The function *IDAAdjInit()* has not been previously called.
- IDA_MEM_FAIL – A memory allocation request has failed.
- IDA_NO_QUAD – Quadrature integration was not activated through a previous call to *IDAQuadInitB()*.
- IDA_ILL_INPUT – The parameter which is an invalid identifier.

Notes: *IDAQuadReInitB()* can be used after a call to either *IDAQuadInitB()* or *IDAQuadInitBS()*.

Backward quadrature extraction function

To extract the values of the quadrature variables at the last return time of *IDASolveB()*, IDAS provides a wrapper for the function *IDAGetQuad()*. The call to this function has the form

int **IDAGetQuadB**(void *ida_mem, int which, *realtype* *tret, *N_Vector* yQB)

The function *IDAGetQuadB()* returns the quadrature solution vector after a successful return from *IDASolveB()*.

Arguments:

- ida_mem – pointer to the IDAS memory.
- tret – the time reached by the solver output.
- which – the identifier of the backward problem.
- yQB – the computed quadrature vector.

Return value:

- IDA_SUCCESS – *IDAGetQuadB()* was successful.
- IDA_MEM_NULL – ida_mem is NULL.
- IDA_NO_ADJ – The function *IDAAdjInit()* has not been previously called.
- IDA_NO_QUAD – Quadrature integration was not initialized.

- IDA_BAD_DKY – yQB was NULL.
- IDA_ILL_INPUT – The parameter which is an invalid identifier.

Notes: To obtain the quadratures associated with a given backward problem at some other time within the last integration step, first obtain a pointer to the proper IDAS memory structure by calling *IDAGetAdjIDABmem()* and then use it to call *IDAGetQuadDky()*.

Warning: The user must allocate space for yQB.

Optional input/output functions for backward quadrature integration

Optional values controlling the backward integration of quadrature equations can be changed from their default values through calls to one of the following functions which are wrappers for the corresponding optional input functions defined in §5.2.4. The user must specify the identifier which of the backward problem for which the optional values are specified.

```
flag = IDASetQuadErrConB(ida_mem, which, errconQ);
flag = IDAQuadSStolerancesB(ida_mem, which, reltolQ, abstolQ);
flag = IDAQuadSVtolerancesB(ida_mem, which, reltolQ, abstolQ);
```

Their return value flag (of type int) can have any of the return values of its counterparts, but it can also be IDA_NO_ADJ if the function *IDAAdjInit()* has not been previously called or IDA_ILL_INPUT if the parameter which was an invalid identifier.

Access to optional outputs related to backward quadrature integration can be obtained by calling the corresponding IDAGetQuad* functions (see §5.2.5). A pointer ida_memB to the IDAS memory block for the backward problem, required as the first argument of these functions, can be obtained through a call to the functions *IDAGetAdjIDABmem()*.

5.5.3 User-supplied functions for adjoint sensitivity analysis

In addition to the required DAE residual function and any optional functions for the forward problem, when using the adjoint sensitivity module in IDAS, the user must supply one function defining the backward problem DAE and, optionally, functions to supply Jacobian-related information and one or two functions that define the preconditioner (if applicable for the choice of SUNLinearSolver object) for the backward problem. Type definitions for all these user-supplied functions are given below.

5.5.3.1 DAE residual for the backward problem

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

typedef int (***IDAResFnB**)(*realtype* t, *N_Vector* y, *N_Vector* yp, *N_Vector* yB, *N_Vector* ypB, *N_Vector* resvalB, void *user_dataB)

This function evaluates the residual of the backward problem DAE system. This could be (2.19) or (2.24).

Arguments:

- t – is the current value of the independent variable.
- y – is the current value of the forward solution vector.
- yp – is the current value of the forward solution derivative vector.
- yB – is the current value of the backward dependent variable vector.

- ypB – is the current value of the backward dependent derivative vector.
- resvalB – is the output vector containing the residual for the backward DAE problem.
- user_dataB – is a pointer to user data, same as passed to IDASetUserDataB() .

Return value: An IDAResFnB should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if an unrecoverabl failure occurred (in which case the integration stops and *IDASolveB()* returns IDA_RESFUNC_FAIL).

Notes: Allocation of memory for resvalB is handled within IDAS. The y, yp, yB, ypB, and resvalB arguments are all of type N_Vector, but yB, ypB, and resvalB typically have different internal representations from y and yp. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each N_Vector implementation). The user_dataB pointer is passed to the user's resB function every time it is called and can be the same as the user_data pointer used for the forward problem.

> **Warning:** Before calling the user's resB function, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the residual function which will halt the integration and *IDASolveB()* will return IDA_RESFUNC_FAIL.

5.5.3.2 DAE residual for the backward problem depending on the forward sensitivities

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

typedef int (***IDAResFnBS**)(*realtype* t, *N_Vector* y, *N_Vector* yp, *N_Vector* *yS, *N_Vector* *ypS, *N_Vector* yB, *N_Vector* ypB, *N_Vector* resvalB, void *user_dataB)

This function evaluates the residual of the backward problem DAE system. This could be (2.19) or (2.24).

Arguments:

- t – is the current value of the independent variable.
- y – is the current value of the forward solution vector.
- yp – is the current value of the forward solution derivative vector.
- yS – a pointer to an array of Ns vectors containing the sensitivities of the forward solution.
- ypS – a pointer to an array of Ns vectors containing the derivatives of the forward sensitivities.
- yB – is the current value of the backward dependent variable vector.
- ypB – is the current value of the backward dependent derivative vector.
- resvalB – is the output vector containing the residual for the backward DAE problem.
- user_dataB – is a pointer to user data, same as passed to IDASetUserDataB() .

Return value: An IDAResFnBS should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if an unrecoverable error occurred (in which case the integration stops and *IDASolveB()* returns IDA_RESFUNC_FAIL).

Notes: Allocation of memory for resvalB is handled within IDAS. The y, yp, yB, ypB, and resvalB arguments are all of type N_Vector, but yB, ypB, and resvalB typically have different internal representations from y and yp. Likewise for each yS[i] and ypS[i]. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each N_Vector implementation). The user_dataB pointer is passed to the user's resBS function every time it is called and can be the same as the user_data pointer used for the forward problem.

Warning: Before calling the user's resBS function, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the residual function which will halt the integration and *IDASolveB()* will return IDA_RESFUNC_FAIL.

5.5.3.3 Quadrature right-hand side for the backward problem

The user must provide an fQB function of type IDAQuadRhsFnB defined by

typedef int (***IDAQuadRhsFnB**)(*realtype* t, *N_Vector* y, *N_Vector* yp, *N_Vector* yB, *N_Vector* ypB, *N_Vector* rhsvalBQ, void *user_dataB)

This function computes the quadrature equation right-hand side for the backward problem.

Arguments:

- t – is the current value of the independent variable.
- y – is the current value of the forward solution vector.
- yp – is the current value of the forward solution derivative vector.
- yB – is the current value of the backward dependent variable vector.
- ypB – is the current value of the backward dependent derivative vector.
- rhsvalBQ – is the output vector containing the residual for the backward quadrature equations.
- user_dataB – is a pointer to user data, same as passed to IDASetUserDataB() .

Return value: An IDAQuadRhsFnB should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and *IDASolveB()* returns IDA_QRHSFUNC_FAIL).

Notes: Allocation of memory for rhsvalBQ is handled within IDAS. The y, yp, yB, ypB, and rhsvalBQ arguments are all of type N_Vector, but they typically all have different internal representations. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each N_Vector implementation). For the sake of computational efficiency, the vector functions in the two N_Vector implementations provided with IDAS do not perform any consistency checks with repsect to their N_Vector arguments (see §6). The user_dataB pointer is passed to the user's fQB function every time it is called and can be the same as the user_data pointer used for the forward problem.

Warning: Before calling the user's fQB function, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the quadrature right-hand side function which will halt the integration and *IDASolveB()* will return IDA_QRHSFUNC_FAIL.

5.5.3.4 Sensitivity-dependent quadrature right-hand side for the backward problem

The user must provide an fQBS function of type IDAQuadRhsFnBS defined by

typedef int (***IDAQuadRhsFnBS**)(*realtype* t, *N_Vector* y, *N_Vector* yp, *N_Vector* *yS, *N_Vector* *ypS, *N_Vector* yB, *N_Vector* ypB, *N_Vector* rhsvalBQS, void *user_dataB)

This function computes the quadrature equation residual for the backward problem.

Arguments:

- t – is the current value of the independent variable.
- y – is the current value of the forward solution vector.
- yp – is the current value of the forward solution derivative vector.
- yS – a pointer to an array of Ns vectors containing the sensitivities of the forward solution.
- ypS – a pointer to an array of Ns vectors containing the derivatives of the forward sensitivities.
- yB – is the current value of the backward dependent variable vector.
- ypB – is the current value of the backward dependent derivative vector.
- rhsvalBQS – is the output vector containing the residual for the backward quadrature equations.
- user_dataB – is a pointer to user data, same as passed to IDASetUserDataB() .

Return value: An IDAQuadRhsFnBS should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and *IDASolveB()* returns IDA_QRHSFUNC_FAIL).

Notes: Allocation of memory for rhsvalBQS is handled within IDAS. The y, yp, yB, ypB, and rhsvalBQS arguments are all of type N_Vector, but they typically do not all have the same internal representations. Likewise for each yS[i] and ypS[i]. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each N_Vector implementation). The user_dataB pointer is passed to the user's fQBS function every time it is called and can be the same as the user_data pointer used for the forward problem.

> **Warning:** Before calling the user's fQBS function, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the quadrature right-hand side function which will halt the integration and *IDASolveB()* will return IDA_QRHSFUNC_FAIL.

5.5.3.5 Jacobian construction for the backward problem (matrix-based linear solvers)

If a matrix-based linear solver module is is used for the backward problem (i.e., *IDASetLinearSolverB()* is called with non-NULL SUNMatrix argument in the step described in §5.5.1), the user may provide a function of type IDALsJacFnB or *IDALsJacFnBS*, defined as follows:

typedef int (***IDALsJacFnB**)(*realtype* tt, *realtype* c_jB, *N_Vector* yy, *N_Vector* yp, *N_Vector* yyB, *N_Vector* ypB, *N_Vector* rrB, *SUNMatrix* JacB, void *user_dataB, *N_Vector* tmp1B, *N_Vector* tmp2B, *N_Vector* tmp3B)

This function computes the Jacobian of the backward problem (or an approximation to it).

Arguments:

- tt – is the current value of the independent variable.
- c_jB – is the scalar in the system Jacobian, proportional to the inverse of the step size (α in (2.6)).
- yy – is the current value of the forward solution vector.

- yp – is the current value of the forward solution derivative vector.
- yB – is the current value of the backward dependent variable vector.
- ypB – is the current value of the backward dependent derivative vector.
- rrB – is the current value of the residual for the backward problem.
- JacB – is the output approximate Jacobian matrix.
- user_dataB – is a pointer to user data — the parameter passed to IDASetUserDataB() .
- tmp1B, tmp2B, tmp3B – are pointers to memory allocated for variables of type N_Vector which can be used by the *IDALsJacFnB* function as temporary storage or work space.

Return value: An *IDALsJacFnB* should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct, while IDALS sets last_flag to IDALS_JACFUNC_RECVR), or a negative value if it failed unrecoverably (in which case the integration is halted, *IDASolveB()* returns IDA_LSETUP_FAIL and IDALS sets last_flag to IDALS_JACFUNC_UNRECVR).

Notes: A user-supplied Jacobian function must load the matrix JacB with an approximation to the Jacobian matrix at the point (tt, yy, yB), where yy is the solution of the original IVP at time tt, and yB is the solution of the backward problem at the same time. 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 $J(t, y)$ is zeroed out prior to calling the user-supplied Jacobian function so only nonzero elements need to be loaded into JacB.

> **Warning:** Before calling the user's IDALsJacFnB, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the Jacobian function which will halt the integration (*IDASolveB()* returns IDA_LSETUP_FAIL and IDALS sets last_flag to IDALS_JACFUNC_UNRECVR).
>
> The previous function type IDADlsJacFnB is identical to IDALsJacFnB, 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.

typedef int (***IDALsJacFnBS**)(*realtype* tt, *realtype* c_jB, *N_Vector* yy, *N_Vector* yp, *N_Vector* *yS, *N_Vector* *ypS, *N_Vector* yyB, *N_Vector* ypB, *N_Vector* rrB, *SUNMatrix* JacB, void *user_dataB, *N_Vector* tmp1B, *N_Vector* tmp2B, *N_Vector* tmp3B);

This function computes the Jacobian of the backward problem (or an approximation to it), in the case where the backward problem depends on the forward sensitivities.

Arguments:

- tt – is the current value of the independent variable.
- c_jB – is the scalar in the system Jacobian, proportional to the inverse of the step size (α in (2.6)).
- yy – is the current value of the forward solution vector.
- yp – is the current value of the forward solution derivative vector.
- yS – a pointer to an array of Ns vectors containing the sensitivities of the forward solution.
- ypS – a pointer to an array of Ns vectors containing the derivatives of the forward solution sensitivities.
- yB – is the current value of the backward dependent variable vector.
- ypB – is the current value of the backward dependent derivative vector.

- rrb – is the current value of the residual for the backward problem.
- JacB – is the output approximate Jacobian matrix.
- user_dataB – is a pointer to user data — the parameter passed to IDASetUserDataB() .
- tmp1B, tmp2B, tmp3B – are pointers to memory allocated for variables of type N_Vector which can be used by *IDALsJacFnBS* as temporary storage or work space.

Return value: An *IDALsJacFnBS* should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct, while IDALS sets last_flag to IDALS_JACFUNC_RECVR), or a negative value if it failed unrecoverably (in which case the integration is halted, *IDASolveB()* returns IDA_LSETUP_FAIL and IDALS sets last_flag to IDALS_JACFUNC_UNRECVR).

Notes: A user-supplied dense Jacobian function must load the matrix JacB with an approximation to the Jacobian matrix at the point (tt, yy, yS, yB), where yy is the solution of the original IVP at time tt, yS is the array of forward sensitivities at time tt, and yB is the solution of the backward problem at the same time. 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 $J(t, y)$ is zeroed out prior to calling the user-supplied Jacobian function so only nonzero elements need to be loaded into JacB.

> **Warning:** Before calling the user's *IDALsJacFnBS*, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the Jacobian function which will halt the integration (*IDASolveB()* returns IDA_LSETUP_FAIL and IDALS sets last_flag to IDALS_JACFUNC_UNRECVR).
>
> The previous function type IDADlsJacFnBS is identical to *IDALsJacFnBS*, 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.

5.5.3.6 Jacobian-vector product for the backward problem (matrix-free linear solvers)

If a matrix-free linear solver is selected for the backward problem (i.e., *IDASetLinearSolverB()* is called with NULL-valued SUNMatrix argument in the steps described in §5.5.1), the user may provide a function of type IDALsJacTimesVecFnB or IDALsJacTimesVecFnBS 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 (***IDALsJacTimesVecFnB**)(*realtype* t, *N_Vector* yy, *N_Vector* yp, *N_Vector* yB, *N_Vector* ypB, *N_Vector* resvalB, *N_Vector* vB, *N_Vector* JvB, *realtype* cjB, void *user_dataB, *N_Vector* tmp1B, *N_Vector* tmp2B)

This function computes the action of the backward problem Jacobian JB on a given vector vB.

Arguments:

- t – is the current value of the independent variable.
- yy – is the current value of the forward solution vector.
- yp – is the current value of the forward solution derivative vector.
- yB – is the current value of the backward dependent variable vector.
- ypB – is the current value of the backward dependent derivative vector.
- resvalB – is the current value of the residual for the backward problem.
- vB – is the vector by which the Jacobian must be multiplied.
- JvB – is the computed output vector, JB*vB .

- cjB – is the scalar in the system Jacobian, proportional to the inverse of the step size (α in (2.6)).
- user_dataB – is a pointer to user data — the same as the user_dataB parameter passed to IDASetUserDataB() .
- tmp1B, tmp2B – are pointers to memory allocated for variables of type N_Vector which can be used by IDALsJacTimesVecFnB as temporary storage or work space.

Return value: The return value of a function of type IDALsJtimesVecFnB should be if successful or nonzero if an error was encountered, in which case the integration is halted.

Notes: A user-supplied Jacobian-vector product function must load the vector JvB with the product of the Jacobian of the backward problem at the point (t, y, yB) and the vector vB. Here, y is the solution of the original IVP at time t and yB is the solution of the backward problem at the same time. The rest of the arguments are equivalent to those passed to a function of type IDALsJacTimesVecFn (see §5.1.5.6). If the backward problem is the adjoint of $\dot{y} = f(t, y)$, then this function is to compute $-(\partial f/\partial y_i)^T v_B$.

Warning: The previous function type IDASpilsJacTimesVecFnB is identical to IDALsJacTimesVecFnB, 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.

typedef int (***IDALsJacTimesVecFnBS**)(*realtype* t, *N_Vector* yy, *N_Vector* yp, *N_Vector* *yyS, *N_Vector* *ypS, *N_Vector* yB, *N_Vector* ypB, *N_Vector* resvalB, *N_Vector* vB, *N_Vector* JvB, *realtype* cjB, void *user_dataB, *N_Vector* tmp1B, *N_Vector* tmp2B)

This function computes the action of the backward problem Jacobian JB on a given vector vB, in the case where the backward problem depends on the forward sensitivities.

Arguments:

- t – is the current value of the independent variable.
- yy – is the current value of the forward solution vector.
- yp – is the current value of the forward solution derivative vector.
- yyS – a pointer to an array of Ns vectors containing the sensitivities of the forward solution.
- ypS – a pointer to an array of Ns vectors containing the derivatives of the forward sensitivities.
- yB – is the current value of the backward dependent variable vector.
- ypB – is the current value of the backward dependent derivative vector.
- resvalB – is the current value of the residual for the backward problem.
- vB – is the vector by which the Jacobian must be multiplied.
- JvB – is the computed output vector, JB*vB .
- cjB – is the scalar in the system Jacobian, proportional to the inverse of the step size (α in (2.6)).
- user_dataB – is a pointer to user data — the same as the user_dataB parameter passed to IDASetUserDataB() .
- tmp1B, tmp2B – are pointers to memory allocated for variables of type N_Vector which can be used by IDALsJacTimesVecFnBS as temporary storage or work space.

Return value: The return value of a function of type IDALsJtimesVecFnBS should be if successful or nonzero if an error was encountered, in which case the integration is halted.

Notes: A user-supplied Jacobian-vector product function must load the vector JvB with the product of the Jacobian of the backward problem at the point (t, y, yB) and the vector vB. Here, y is the solution of the original

IVP at time t and yB is the solution of the backward problem at the same time. The rest of the arguments are equivalent to those passed to a function of type IDALsJacTimesVecFn (see §5.1.5.6).

Warning: The previous function type IDASpilsJacTimesVecFnBS is identical to IDALsJacTimesVecFnBS, 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.

5.5.3.7 Jacobian-vector product setup for the backward problem (matrix-free linear solvers)

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

typedef int (***IDALsJacTimesSetupFnB**)(*realtype* tt, *N_Vector* yy, *N_Vector* yp, *N_Vector* yB, *N_Vector* ypB, *N_Vector* resvalB, *realtype* cjB, void *user_dataB)

This function preprocesses and/or evaluates Jacobian data needed by the Jacobian-times-vector routine for the backward problem.

Arguments:

- tt – is the current value of the independent variable.
- yy – is the current value of the dependent variable vector, $y(t)$.
- yp – is the current value of $\dot{y}(t)$.
- yB – is the current value of the backward dependent variable vector.
- ypB – is the current value of the backward dependent derivative vector.
- resvalB – is the current value of the residual for the backward problem.
- cjB – is the scalar in the system Jacobian, proportional to the inverse of the step size (α in (2.6)).
- user_dataB – is a pointer to user data — the same as the user_dataB parameter passed to IDASetUserDataB() .

Return value: The value returned by the Jacobian-vector setup function should be 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: Each call to the Jacobian-vector setup function is preceded by a call to the backward problem residual user function with the same (t,y, yp, yB, ypB) arguments. Thus, the setup function can use any auxiliary data that is computed and saved during the evaluation of the DAE residual. If the user's IDALsJacTimesVecFnB function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, the user will need to add a pointer to ida_mem to user_dataB and then use the IDAGet* functions described in §5.1.4.12. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundials_types.h.

Warning: The previous function type IDASpilsJacTimesSetupFnB is identical to IDALsJacTimesSetupFnB, 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.

typedef int (***IDALsJacTimesSetupFnBS**)(*realtype* tt, *N_Vector* yy, *N_Vector* yp, *N_Vector* *yyS, *N_Vector* *ypS, *N_Vector* yB, *N_Vector* ypB, *N_Vector* resvalB, *realtype* cjB, void *user_dataB)

This function preprocesses and/or evaluates Jacobian data needed by the Jacobian-times-vector routine for the backward problem, in the case that the backward problem depends on the forward sensitivities.

Arguments:

- `tt` – is the current value of the independent variable.
- `yy` – is the current value of the dependent variable vector, $y(t)$.
- `yp` – is the current value of $\dot{y}(t)$.
- `yyS` – a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution.
- `ypS` – a pointer to an array of `Ns` vectors containing the derivatives of the forward sensitivities.
- `yB` – is the current value of the backward dependent variable vector.
- `ypB` – is the current value of the backward dependent derivative vector.
- `resvalB` – is the current value of the residual for the backward problem.
- `cjB` – is the scalar in the system Jacobian, proportional to the inverse of the step size (α in (2.6)).
- `user_dataB` – is a pointer to user data — the same as the `user_dataB` parameter passed to `IDASetUserDataB()` .

Return value: The value returned by the Jacobian-vector setup function should be 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: Each call to the Jacobian-vector setup function is preceded by a call to the backward problem residual user function with the same `(t,y, yp, yyS, ypS, yB, ypB)` arguments. Thus, the setup function can use any auxiliary data that is computed and saved during the evaluation of the DAE residual. If the user's `IDALsJacTimesVecFnB` function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current stepsize, the error weights, etc. To obtain these, the user will need to add a pointer to `ida_mem` to `user_dataB` and then use the `IDAGet*` functions described in §5.5.2.11. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`. The previous function type `IDASpilsJacTimesSetupFnBS` is deprecated.

Warning: The previous function type `IDASpilsJacTimesSetupFnBS` is identical to *IDALsJacTimesSetupFnBS*, 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.

5.5.3.8 Preconditioner solve for the backward problem (iterative linear solvers)

If preconditioning is used during integration of the backward problem, then the user must provide a function to solve the linear system $Pz = r$, where P is a left preconditioner matrix. This function must have one of the following two forms:

typedef int (***IDALsPrecSolveFnB**)(*realtype* t, *N_Vector* yy, *N_Vector* yp, *N_Vector* yB, *N_Vector* ypB, *N_Vector* resvalB, *N_Vector* rvecB, *N_Vector* zvecB, *realtype* cjB, *realtype* deltaB, void *user_dataB)

This function solves the preconditioning system $Pz = r$ for the backward problem.

Arguments:

- `t` – is the current value of the independent variable.
- `yy` – is the current value of the forward solution vector.
- `yp` – is the current value of the forward solution derivative vector.
- `yB` – is the current value of the backward dependent variable vector.
- `ypB` – is the current value of the backward dependent derivative vector.

- resvalB – is the current value of the residual for the backward problem.
- rvecB – is the right-hand side vector r of the linear system to be solved.
- zvecB – is the computed output vector.
- cjB – is the scalar in the system Jacobian, proportional to the inverse of the step size (α in (2.6)).
- deltaB – is an input tolerance to be used if an iterative method is employed in the solution.
- user_dataB – is a pointer to user data — the same as the user_dataB parameter passed to the function IDASetUserDataB() .

Return value: The return value of a preconditioner solve function for the backward problem should be 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).

Warning: The previous function type IDASpilsPrecSolveFnB is identical to IDALsPrecSolveFnB, and is deprecated.

typedef int (***IDALsPrecSolveFnBS**)(*realtype* t, *N_Vector* yy, *N_Vector* yp, *N_Vector* *yyS, *N_Vector* *ypS, *N_Vector* yB, *N_Vector* ypB, *N_Vector* resvalB, *N_Vector* rvecB, *N_Vector* zvecB, *realtype* cjB, *realtype* deltaB, void *user_dataB)

This function solves the preconditioning system $Pz = r$ for the backward problem, for the case in which the backward problem depends on the forward sensitivities.

Arguments:

- t – is the current value of the independent variable.
- yy – is the current value of the forward solution vector.
- yp – is the current value of the forward solution derivative vector.
- yyS – a pointer to an array of Ns vectors containing the sensitivities of the forward solution.
- ypS – a pointer to an array of Ns vectors containing the derivatives of the forward sensitivities.
- yB – is the current value of the backward dependent variable vector.
- ypB – is the current value of the backward dependent derivative vector.
- resvalB – is the current value of the residual for the backward problem.
- rvecB – is the right-hand side vector r of the linear system to be solved.
- zvecB – is the computed output vector.
- cjB – is the scalar in the system Jacobian, proportional to the inverse of the step size (α in (2.6)).
- deltaB – is an input tolerance to be used if an iterative method is employed in the solution.
- user_dataB – is a pointer to user data — the same as the user_dataB parameter passed to the function IDASetUserDataB() .

Return value: The return value of a preconditioner solve function for the backward problem should be 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).

Warning: The previous function type IDASpilsPrecSolveFnBS is identical to IDALsPrecSolveFnBS, and is deprecated.

5.5.3.9 Preconditioner setup for the backward problem (iterative linear solvers)

If the user's preconditioner requires that any Jacobian-related data be preprocessed or evaluated, then this needs to be done in a user-supplied function of one of the following two types:

typedef int (***IDALsPrecSetupFnB**)(*realtype* t, *N_Vector* yy, *N_Vector* yp, *N_Vector* yB, *N_Vector* ypB, *N_Vector* resvalB, *realtype* cjB, void *user_dataB)

This function preprocesses and/or evaluates Jacobian-related data needed by the preconditioner for the backward problem.

Arguments:

- `t` – is the current value of the independent variable.
- `yy` – is the current value of the forward solution vector.
- `yp` – is the current value of the forward solution vector.
- `yB` – is the current value of the backward dependent variable vector.
- `ypB` – is the current value of the backward dependent derivative vector.
- `resvalB` – is the current value of the residual for the backward problem.
- `cjB` – is the scalar in the system Jacobian, proportional to the inverse of the step size (α in (2.6)).
- `user_dataB` – is a pointer to user data — the same as the `user_dataB` parameter passed to the function `IDASetUserDataB()` .

Return value: The return value of a preconditioner setup function for the backward problem should be 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).

Warning: The previous function type `IDASpilsPrecSetupFnB` is identical to `IDALsPrecSetupFnB`, and is deprecated.

typedef int (***IDALsPrecSetupFnBS**)(*realtype* t, *N_Vector* yy, *N_Vector* yp, *N_Vector* *yyS, *N_Vector* *ypS, *N_Vector* yB, *N_Vector* ypB, *N_Vector* resvalB, *realtype* cjB, void *user_dataB)

This function preprocesses and/or evaluates Jacobian-related data needed by the preconditioner for the backward problem, in the case where the backward problem depends on the forward sensitivities.

Arguments:

- `t` – is the current value of the independent variable.
- `yy` – is the current value of the forward solution vector.
- `yp` – is the current value of the forward solution vector.
- `yyS` – a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution.
- `ypS` – a pointer to an array of `Ns` vectors containing the derivatives of the forward sensitivities.
- `yB` – is the current value of the backward dependent variable vector.
- `ypB` – is the current value of the backward dependent derivative vector.
- `resvalB` – is the current value of the residual for the backward problem.
- `cjB` – is the scalar in the system Jacobian, proportional to the inverse of the step size (α in (2.6)).
- `user_dataB` – is a pointer to user data — the same as the `user_dataB` parameter passed to the function `IDASetUserDataB()` .

Return value: The return value of a preconditioner setup function for the backward problem should be 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).

Warning: The previous function type `IDASpilsPrecSetupFnBS` is identical to `IDALsPrecSetupFnBS`, and is deprecated.

5.5.4 Using the band-block-diagonal preconditioner for backward problems

As on the forward integration phase, the efficiency of Krylov iterative methods for the solution of linear systems can be greatly enhanced through preconditioning. The band-block-diagonal preconditioner module IDABBDPRE, provides interface functions through which it can be used on the backward integration phase.

The adjoint module in IDAS offers an interface to the band-block-diagonal preconditioner module IDABBDPRE described in section §5.3.1. This generates a preconditioner that is a block-diagonal matrix with each block being a band matrix and can be used with one of the Krylov linear solvers and with the MPI-parallel vector module `NVECTOR_PARALLEL`.

In order to use the IDABBDPRE module in the solution of the backward problem, the user must define one or two additional functions, described at the end of this section.

5.5.4.1 Usage of IDABBDPRE for the backward problem

The IDABBDPRE module is initialized by calling the following function, *after* an iterative linear solver for the backward problem has been attached to IDAS by calling *IDASetLinearSolverB()* (see §5.5.2.6).

int **IDABBDPrecInitB**(void *ida_mem, int which, *sunindextype* NlocalB, *sunindextype* mudqB, *sunindextype* mldqB, *sunindextype* mukeepB, *sunindextype* mlkeepB, *realtype* dqrelyB, *IDABBDLocalFnB* GresB, *IDABBDCommFnB* GcommB)

The function *IDABBDPrecInitB()* initializes and allocates memory for the IDABBDPRE preconditioner for the backward problem.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `which` – the identifier of the backward problem.
- `NlocalB` – local vector dimension for the backward problem.
- `mudqB` – upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
- `mldqB` – lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
- `mukeepB` – upper half-bandwidth of the retained banded approximate Jacobian block.
- `mlkeepB` – lower half-bandwidth of the retained banded approximate Jacobian block.
- `dqrelyB` – the relative increment in components of `yB` used in the difference quotient approximations. The default is `dqrelyB` $= \sqrt{\text{unit roundoff}}$, which can be specified by passing `dqrely` $= 0.0$.
- `GresB` – the C function which computes $G_B(t, y, \dot{y}, y_B, \dot{y}_B)$, the function approximating the residual of the backward problem.
- `GcommB` – the optional C function which performs all interprocess communication required for the computation of G_B.

Return value:

- IDALS_SUCCESS – The call to *IDABBDPrecInitB()* was successful.
- IDALS_MEM_FAIL – A memory allocation request has failed.
- IDALS_MEM_NULL – The ida_mem argument was NULL.
- IDALS_LMEM_NULL – No linear solver has been attached.
- IDALS_ILL_INPUT – An invalid parameter has been passed.

To reinitialize the IDABBDPRE preconditioner module for the backward problem, possibly with a change in mudqB, mldqB, or dqrelyB, call the following function:

int **IDABBDPrecReInitB**(void *ida_mem, int which, *sunindextype* mudqB, *sunindextype* mldqB, *realtype* dqrelyB)
The function *IDABBDPrecReInitB()* reinitializes the IDABBDPRE preconditioner for the backward problem.

Arguments:

- ida_mem – pointer to the IDAS memory block returned by *IDACreate()*.
- which – the identifier of the backward problem.
- mudqB – upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
- mldqB – lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
- dqrelyB – the relative increment in components of yB used in the difference quotient approximations.

Return value:

- IDALS_SUCCESS – The call to *IDABBDPrecReInitB()* was successful.
- IDALS_MEM_FAIL – A memory allocation request has failed.
- IDALS_MEM_NULL – The ida_mem argument was NULL.
- IDALS_PMEM_NULL – The *IDABBDPrecInitB()* has not been previously called.
- IDALS_LMEM_NULL – No linear solver has been attached.
- IDALS_ILL_INPUT – An invalid parameter has been passed.

5.5.4.2 User-supplied functions for IDABBDPRE

To use the IDABBDPRE module, the user must supply one or two functions which the module calls to construct the preconditioner: a required function GresB (of type *IDABBDLocalFnB*) which approximates the residual of the backward problem and which is computed locally, and an optional function GcommB (of type *IDABBDCommFnB*) which performs all interprocess communication necessary to evaluate this approximate residual (see §5.3.1). The prototypes for these two functions are described below.

typedef int (***IDABBDLocalFnB**)(*sunindextype* NlocalB, *realtype* t, *N_Vector* y, *N_Vector* yp, *N_Vector* yB, *N_Vector* ypB, *N_Vector* gB, void *user_dataB)
This GresB function loads the vector gB, an approximation to the residual of the backward problem, as a function of t, y, yp, and yB and ypB.

Arguments:

- NlocalB – is the local vector length for the backward problem.
- t – is the value of the independent variable.
- y – is the current value of the forward solution vector.
- yp – is the current value of the forward solution derivative vector.
- yB – is the current value of the backward dependent variable vector.

- ypB – is the current value of the backward dependent derivative vector.
- gB – is the output vector, $G_B(t, y, \dot{y}, y_B, \dot{y}_B)$.
- user_dataB – is a pointer to user data — the same as the user_dataB parameter passed to IDASetUserDataB() .

Return value: An IDABBDLocalFnB should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and *IDASolveB()* returns IDA_LSETUP_FAIL).

Notes: This routine must assume that all interprocess communication of data needed to calculate gB has already been done, and this data is accessible within user_dataB.

> **Warning:** Before calling the user's IDABBDLocalFnB, IDAS needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, IDAS triggers an unrecoverable failure in the preconditioner setup function which will halt the integration (*IDASolveB()* returns IDA_LSETUP_FAIL).

typedef int (***IDABBDCommFnB**)(*sunindextype* NlocalB, *realtype* t, *N_Vector* y, *N_Vector* yp, *N_Vector* yB, *N_Vector* ypB, void *user_dataB)

This GcommB function performs all interprocess communications necessary for the execution of the GresB function above, using the input vectors y, yp, yB and ypB.

Arguments:

- NlocalB – is the local vector length.
- t – is the value of the independent variable.
- y – is the current value of the forward solution vector.
- yp – is the current value of the forward solution derivative vector.
- yB – is the current value of the backward dependent variable vector.
- ypB – is the current value of the backward dependent derivative vector.
- user_dataB – is a pointer to user data — the same as the user_dataB parameter passed to IDASetUserDataB() .

Return value: An IDABBDCommFnB should return 0 if successful, a positive value if a recoverable error occurred (in which case IDAS will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and *IDASolveB()* returns IDA_LSETUP_FAIL).

Notes: The GcommB function is expected to save communicated data in space defined within the structure user_dataB.

Each call to the GcommB function is preceded by a call to the function that evaluates the residual of the backward problem with the same t, y, yp, yB and ypB arguments. If there is no additional communication needed, then pass GcommB = NULL to *IDABBDPrecInitB()*.

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.

Similarly, although both the *SUNNonlinearSolver_Newton* and *SUNNonlinearSolver_FixedPoint* modules may be used with any of the IVP solvers (CVODE(S), IDA(S) and ARKODE) for complex-valued problems, the Anderson-acceleration option with SUNNonlinearSolver_FixedPoint cannot be used due to its reliance on *N_VDotProd()*. By this same logic, the Anderson acceleration feature within KINSOL will also not work with complex-valued vectors.

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

$$H(\alpha) = \begin{cases} 1 & \alpha > 0 \\ 0 & \alpha \le 0 \end{cases}$$

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:

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

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 IDAS

In Table 6.2 below, we list the vector functions used in the `N_Vector` module used by the IDAS package. The table also shows, for each function, which of the code modules uses the function. The IDAS column shows function usage within the main integrator module, while the remaining columns show function usage within the IDALS linear solvers interface, and the IDABBDPRE preconditioner module.

At this point, we should emphasize that the IDAS user does not need to know anything about the usage of vector functions by the IDAS code modules in order to use IDAS. The information is presented as an implementation detail for the interested reader.

Table 6.2: List of vector functions usage by IDAS code modules

	IDAS	IDALS	IDABBDPRE	IDAA
N_VGetVectorID()				
N_VGetLength()	4			
N_VClone()	x	x	x	x
N_VCloneEmpty()	1			
N_VDestroy()	x	x	x	x
N_VCloneVectorArray()	x	x		
N_VDestroyVectorArray()	x	x		
N_VSpace()	x	2		
N_VGetArrayPointer()	1	x		
N_VSetArrayPointer()	1			
N_VLinearSum()	x	x	x	
N_VConst()	x	x	x	
N_VProd()	x			
N_VDiv()	x			
N_VScale()	x	x	x	x
N_VAbs()	x			
N_VInv()	x			
N_VAddConst()	x			
N_VMaxNorm()	x			
N_VWrmsNorm()	x	x		
N_VMin()	x			
N_VMinQuotient()	x			
N_VConstrMask()	x			
N_VWrmsNormMask()	x			
N_VCompare()	x			
N_VLinearCombination()	x			
N_VScaleAddMulti()	x			
N_VDotProdMulti()	3			
N_VLinearSumVectorArray()	x			
N_VScaleVectorArray()	x			
N_VConstVectorArray()	x			
N_VWrmsNormVectorArray()	x			
N_VWrmsNormMaskVectorArray()	x			
N_VScaleAddMultiVectorArray()	x			
N_VLinearCombinationVectorArray()	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 IDAS modules for user feedback.
3. The optional function `N_VDotProdMulti` is only used when Classical Gram-Schmidt is enabled with SPGMR or SPFGMR. The remaining operations from Tables §6.2.2 and §6.2.3 not listed above are unused and a user-supplied `N_Vector` module for IDAS could omit these operations.
4. This routine is only used when an iterative or matrix iterative `SUNLinearSolver` module is supplied to IDAS.

Of the functions listed in §6.2, *N_VDotProd()* `N_VWL2Norm()`, *N_VL1Norm()*, *N_VInvTest()*, and *N_VGetCommunicator()* are *not* used by IDAS. Therefore a user-supplied `N_Vector` module for IDAS could omit these functions (although some may be needed by `SUNNonlinearSolver` or `SUNLinearSolver` modules).

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 §11. 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 §11. 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 §11.

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 §11.

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_exec;
  booleantype        own_helper;
  SUNMemory          host_data;
  SUNMemory          device_data;
  SUNCudaExecPolicy* stream_exec_policy;
  SUNCudaExecPolicy* reduce_exec_policy;
  SUNMemoryHelper    mem_helper;
  void*              priv; /* 'private' data */
};

typedef struct _N_VectorContent_Cuda *N_VectorContent_Cuda;
```

The content members are the vector length (size), 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 [33].

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 *SUNCudaBlockReduceExecPolicy()*. Any custom execution policy for reductions must ensure that the grid dimensions (number of thread blocks) is a multiple of the CUDA warp size (32). See §6.10.2 below for more information about the *SUNCudaExecPolicy* class.

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:
   virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const = 0;
   virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const = 0;
   virtual cudaStream_t stream() const = 0;
   virtual ExecPolicy* clone() const = 0;
   virtual ~ExecPolicy() {}
};
```

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, const cudaStream_t stream = 0)
      : blockDim_(blockDim), stream_(stream)
   {}

   ThreadDirectExecPolicy(const ThreadDirectExecPolicy& ex)
      : blockDim_(ex.blockDim_), stream_(ex.stream_)
   {}

   virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const
   {
      return (numWorkUnits + blockSize() - 1) / blockSize();
   }

   virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const
   {
      return blockDim_;
   }
```

(continues on next page)

(continued from previous page)

```
   virtual cudaStream_t stream() const
   {
      return stream_;
   }

   virtual ExecPolicy* clone() const
   {
      return static_cast<ExecPolicy*>(new ThreadDirectExecPolicy(*this));
   }

private:
   const cudaStream_t stream_;
   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.

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 [49]. 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_data;
  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.

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:
   virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const = 0;
   virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const = 0;
   virtual hipStream_t stream() const = 0;
   virtual ExecPolicy* clone() const = 0;
   virtual ~ExecPolicy() {}
};
```

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, const hipStream_t stream = 0)
      : blockDim_(blockDim), stream_(stream)
   {}

   ThreadDirectExecPolicy(const ThreadDirectExecPolicy& ex)
      : blockDim_(ex.blockDim_), stream_(ex.stream_)
   {}

   virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const
   {
      return (numWorkUnits + blockSize() - 1) / blockSize();
   }

   virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const
   {
      return blockDim_;
   }

   virtual hipStream_t stream() const
   {
      return stream_;
   }

   virtual ExecPolicy* clone() const
   {
```

(continues on next page)

(continued from previous page)

```
    return static_cast<ExecPolicy*>(new ThreadDirectExecPolicy(*this));
  }

private:
  const hipStream_t stream_;
  const size_t blockDim_;
};
```

In total, SUNDIALS provides 3 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.

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
{
```

(continues on next page)

(continued from previous page)

```
   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 [33].

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;
   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
   {
      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 SUNSyclBlockReduce-

ExecPolicy 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 §4.5.2.5, 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 §4.5.2.5, 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 [46]. 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 §4.2)

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 §4.2)

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 §4.2)

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 §4.2)

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:

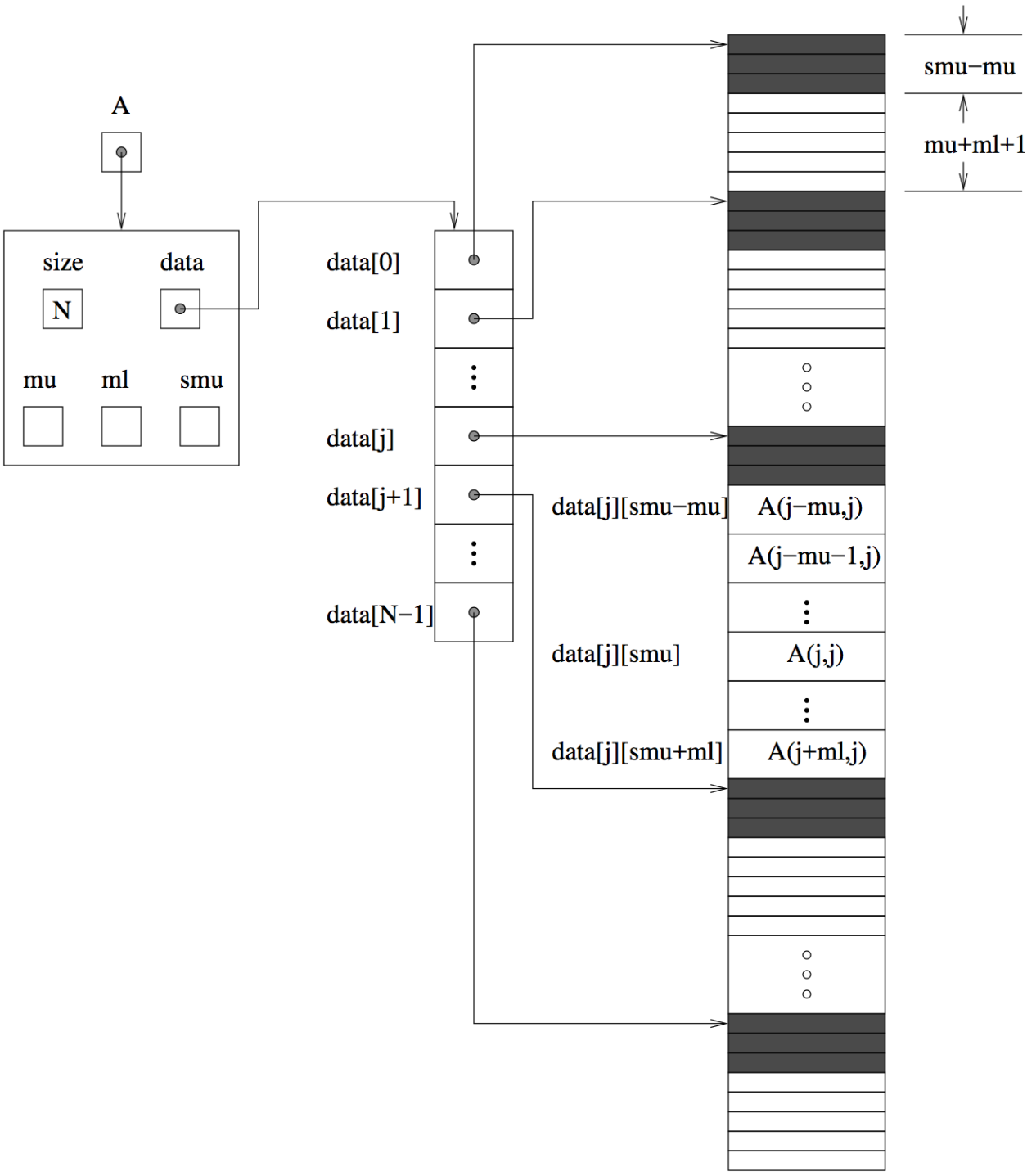

Fig. 7.1: Diagram of the storage for the SUNMATRIX_BAND module. Here `A` is an $N \times N$ band matrix with upper and lower half-bandwidths `mu` and `ml`, respectively. The rows and columns of `A` are numbered from 0 to `N-1` and the (i, j)-th element of `A` is denoted `A(i,j)`. The greyed out areas 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 [54]. 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 )
  ```

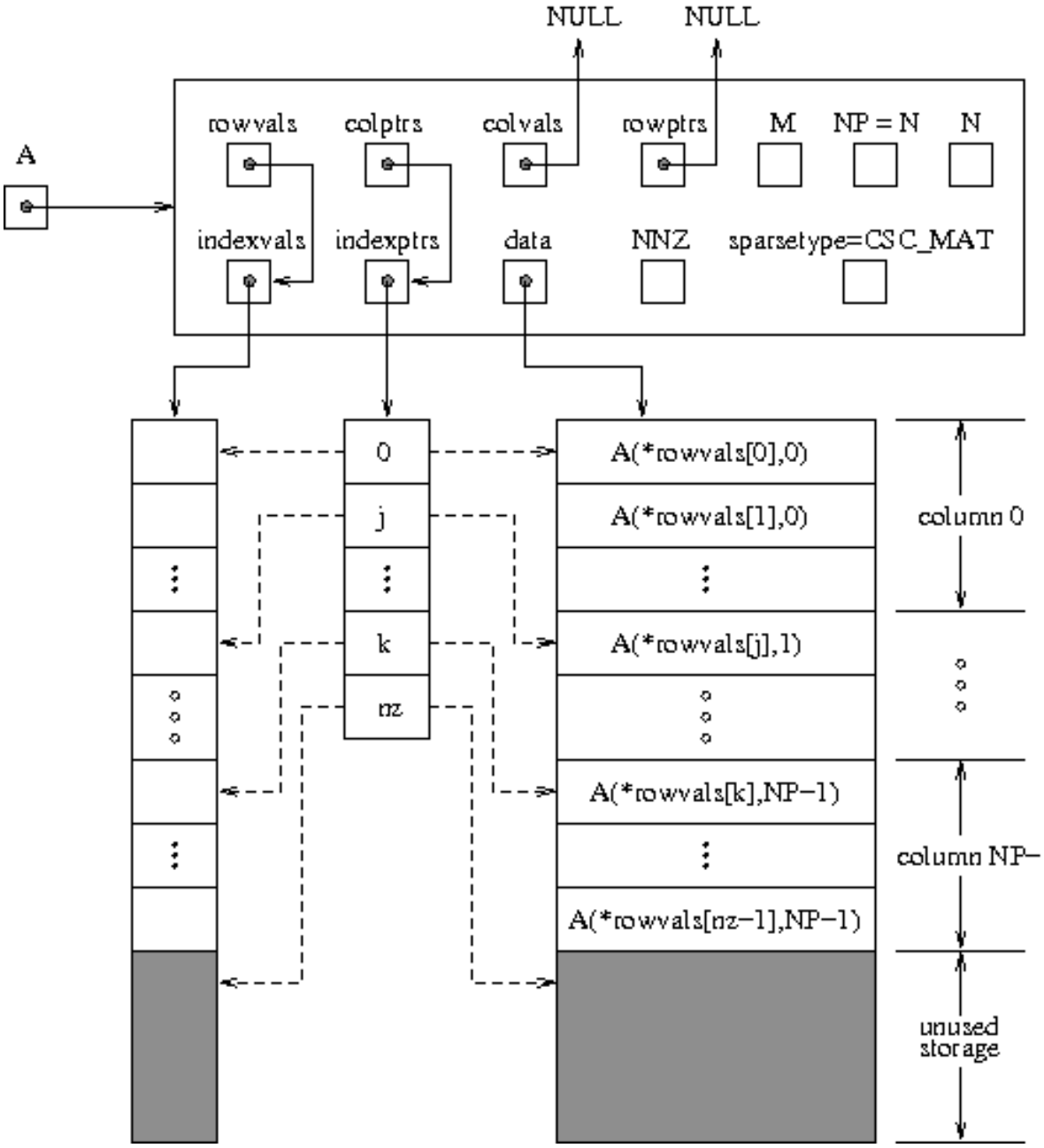

Fig. 7.2: Diagram of the storage for a compressed-sparse-column matrix of type SUNMATRIX_SPARSE: Here `A` is an $M \times N$ sparse CSC matrix with storage for up to `NNZ` nonzero entries (the allocated length of both `data` and `indexvals`). The entries in `indexvals` may assume values from `0` to `M-1`, 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 [26, 39, 40, 55]. 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 IDAS

In Table 7.2, we list the matrix functions in the `SUNMatrix` module used within the IDAS package. The table also shows, for each function, which of the code modules uses the function. The main IDAS integrator does not call any `SUNMatrix` functions directly, so the table columns are specific to the IDALS and IDABBDPRE preconditioner modules. We further note that the IDALS interface only utilizes these routines when supplied with a *matrix-based* linear solver, i.e., the `SUNMatrix` object passed to *IDASetLinearSolver()* was not `NULL`.

At this point, we should emphasize that the IDAS user does not need to know anything about the usage of matrix functions by the IDAS code modules in order to use IDAS. The information is presented as an implementation detail for the interested reader.

Table 7.2: List of matrix functions usage by IDAS code modules

	IDALS	IDABBDPRE
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 IDAS are: *SUNMatCopy()*, *SUNMatClone()*, *SUNMatScaleAdd()*, *SUNMatScaleAddI()* and *SUNMatMatvec()*. Therefore a user-supplied `SUNMatrix` module for IDAS could omit these functions.

We note that the IDABBDPRE 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 IDALS 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 IDAS SUNLinearSolver interface

Table 8.3 below lists the `SUNLinearSolver` module linear solver functions used within the IDALS interface. As with the `SUNMatrix` module, we emphasize that the IDA user does not need to know detailed usage of linear solver functions by the IDA code modules in order to use IDA. 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. Although IDALS does not call `SUNLinSolLastFlag` directly, this routine is available for users to query linear solver issues directly.
2. Although IDALS 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 IDALS interface

	DIRECT	ITERATIVE	MATRIX_ITERATIVE
SUNLinSolGetType()	x	x	x
SUNLinSolSetATimes()	†	x	†
SUNLinSolSetPreconditioner()	†	†	†
SUNLinSolSetScalingVectors()	†	†	†
SUNLinSolInitialize()	x	x	x
SUNLinSolSetup()	x	x	x
SUNLinSolSolve()	x	x	x
SUNLinSolNumIters()		x	x
SUNLinSolResid()		x	x
[1] *SUNLinSolLastFlag()*			
[2] *SUNLinSolFree()*			
SUNLinSolSpace()	†	†	†

Since there are a wide range of potential `SUNLinearSolver` use cases, the following subsections describe some details of the IDALS 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. IDALS will update the matrix information infrequently according to the strategies outlined in §2. To this end, we differentiate between the *desired* linear system $Jx = b$ with $J = \left(\dfrac{\partial F}{\partial y} - c_j \dfrac{\partial F}{\partial \dot{y}}\right)$, and the *actual* linear system $\bar{J}\bar{x} = b$ with

$$\bar{J} = \frac{\partial \bar{F}}{\partial y} - \bar{c}_j \frac{\partial \bar{F}}{\partial \dot{y}},$$

where the overlines indicate the lagged versions of these numbers and matrices.

Since IDALS updates the `SUNMatrix` objects infrequently and it is likely that $c_j \neq \bar{c}_j$, then typically $J \neq \bar{J}$. Thus after calling the `SUNLinearSolver`-provided `SUNLinSolSolve` routine, we test whether $\dfrac{c_j}{\bar{c}_j} \neq 1$, and if this is the case we scale the solution $\bar{x}$ to correct the linear system solution x via

$$x = \frac{2}{1 + c_j/\bar{c}_j}\bar{x}. \tag{8.3}$$

The motivation for this selection of the scaling factor $c = 2/(1 + c_j/\bar{c}_j)$ is discussed in detail in [6, 29]. In short, if we consider a stationary iteration for the linear system as consisting of a solve with $\bar{J}$ followed by scaling by c, then for a linear constant-coefficient problem, the error in the solution vector will be reduced at each iteration by the error matrix $E = I - c\bar{J}^{-1}J$, with a convergence rate given by the spectral radius of E. Assuming that stiff systems have a spectrum spread widely over the left half-plane, c is chosen to minimize the magnitude of the eigenvalues of E.

8.2.2 Iterative linear solver tolerance

If the SUNLinearSolver object self-identifies as having type SUNLINEARSOLVER_ITERATIVE or SUNLINEARSOLVER_MATRIX_ITERATIVE then IDALS will set the input tolerance delta as described in §2.1. However, if the iterative linear solver does not support scaling matrices (i.e., the SUNLinSolSetScalingVectors routine is NULL), then IDALS 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 solution components have similar magnitude; hence the error weight vector W used in the WRMS norm (see §2.1) should satisfy the assumption

$$W_i \approx W_{mean}, \quad \text{for} \quad i = 0, \ldots, n-1.$$

2. The SUNLinearSolver object uses a standard 2-norm to measure convergence.

Since IDA uses identical left and right scaling matrices, $S_1 = S_2 = S = \mathrm{diag}(W)$, then the linear solver convergence requirement is converted as follows (using the notation from equations (8.1) – (8.2)):

$$\begin{aligned}
& \left\| \tilde{b} - \tilde{A}\tilde{x} \right\|_2 < \text{tol} \\
\Leftrightarrow \quad & \left\| SP_1^{-1}b - SP_1^{-1}Ax \right\|_2 < \text{tol} \\
\Leftrightarrow \quad & \sum_{i=0}^{n-1} \left[W_i \left(P_1^{-1}(b - Ax) \right)_i \right]^2 < \text{tol}^2 \\
\Leftrightarrow \quad & W_{mean}^2 \sum_{i=0}^{n-1} \left[\left(P_1^{-1}(b - Ax) \right)_i \right]^2 < \text{tol}^2 \\
\Leftrightarrow \quad & \sum_{i=0}^{n-1} \left[\left(P_1^{-1}(b - Ax) \right)_i \right]^2 < \left(\frac{\text{tol}}{W_{mean}} \right)^2 \\
\Leftrightarrow \quad & \left\| P_1^{-1}(b - Ax) \right\|_2 < \frac{\text{tol}}{W_{mean}}
\end{aligned}$$

Therefore the tolerance scaling factor

$$W_{mean} = \|W\|_2 / \sqrt{n}$$

is computed and the scaled tolerance delta$= \text{tol}/W_{mean}$ is supplied to the SUNLinearSolver object.

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 §4.2)

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 §4.2)

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 §4.2)

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 ([18, 51]). 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 §11.1.4 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 §4.2)

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 §4.1 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 §11.1.4 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 §4.2)

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 §4.1 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 §11.1.4 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 [46]. 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 §4.2)

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 §4.2)

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 [27]) 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

$$\tilde{A}\tilde{x} = \tilde{b} \tag{8.4}$$

where

$$\begin{aligned} \tilde{A} &= SP^{-1}AP^{-1}S, \\ \tilde{b} &= SP^{-1}b, \\ \tilde{x} &= S^{-1}Px. \end{aligned} \tag{8.5}$$

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 §4.2)

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 §11.1.2 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 §11.1.2 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 [47] 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 §4.2)

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 §11.1.2 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 §11.1.2 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;
  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 [44] 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 §4.2)

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 §11.1.2 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 §11.1.2 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 $(\text{maxl}+1) \times \text{maxl}$ Hessenberg matrix. It is stored row-wise so that the (i,j)th element is given by `Hes[i][j]`,
- `givens` - a length $2\,\text{maxl}$ array which represents the Givens rotation matrices that arise in the FGMRES algorithm. These matrices are $F_0, F_1, \ldots, F_j$, where

$$F_i = \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_i & -s_i & & & \\ & & & s_i & c_i & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix},$$

  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 [45] 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 §11.1.2 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 §11.1.2 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 [25] 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 §4.2)

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 §11.1.2 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 §11.1.2 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;
  int last_flag;
  SUNATimesFn ATimes;
  void* ATData;
  SUNPSetupFn Psetup;
  SUNPSolveFn Psolve;
```

(continues on next page)

(continued from previous page)

```
  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 §4.2)

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 [26, 39, 40, 55]. 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 §11.1.4 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 §4.2)

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 [20, 38, 56]. 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 §11.1.4 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 §4.1 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 [53]. 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 §4.2)

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

Nonlinear Algebraic Solvers

SUNDIALS time integration packages are written in terms of generic nonlinear solver operations defined by the SUNNonlinSol API and implemented by a particular SUNNonlinSol module of type `SUNNonlinearSolver`. Users can supply their own SUNNonlinSol module, or use one of the modules provided with SUNDIALS. Depending on the package, nonlinear solver modules can either target system presented in a rootfinding ($F(y) = 0$) or fixed-point ($G(y) = y$) formulation. For more information on the formulation of the nonlinear system(s) see the §9.2 section.

The time integrators in SUNDIALS specify a default nonlinear solver module and as such this chapter is intended for users that wish to use a non-default nonlinear solver module or would like to provide their own nonlinear solver implementation. Users interested in using a non-default solver module may skip the description of the SUNNonlinSol API in section §9.1 and proceeded to the subsequent sections in this chapter that describe the SUNNonlinSol modules provided with SUNDIALS.

For users interested in providing their own SUNNonlinSol module, the following section presents the SUNNonlinSol API and its implementation beginning with the definition of SUNNonlinSol functions in the sections §9.1.1, §9.1.2 and §9.1.3. This is followed by the definition of functions supplied to a nonlinear solver implementation in the section §9.1.4. The nonlinear solver return codes are given in the section §9.1.5. The `SUNNonlinearSolver` type and the generic SUNNonlinSol module are defined in the section §9.1.6. Finally, the section §9.1.7 lists the requirements for supplying a custom SUNNonlinSol module. Users wishing to supply their own SUNNonlinSol module are encouraged to use the SUNNonlinSol implementations provided with SUNDIALS as a template for supplying custom nonlinear solver modules.

9.1 The SUNNonlinearSolver API

The SUNNonlinSol API defines several nonlinear solver operations that enable SUNDIALS integrators to utilize any SUNNonlinSol implementation that provides the required functions. These functions can be divided into three categories. The first are the core nonlinear solver functions. The second consists of "set" routines to supply the nonlinear solver with functions provided by the SUNDIALS time integrators and to modify solver parameters. The final group consists of "get" routines for retrieving nonlinear solver statistics. All of these functions are defined in the header file `sundials/sundials_nonlinearsolver.h`.

9.1.1 SUNNonlinearSolver core functions

The core nonlinear solver functions consist of two required functions to get the nonlinear solver type (`SUNNonlinsSolGetType()`) and solve the nonlinear system (*SUNNonlinSolSolve()*). The remaining three functions for nonlinear solver initialization (`SUNNonlinSolInitialization()`), setup (*SUNNonlinSolSetup()*), and destruction (*SUNNonlinSolFree()*) are optional.

SUNNonlinearSolver_Type **SUNNonlinSolGetType**(*SUNNonlinearSolver* NLS)
This *required* function returns the nonlinear solver type.

Arguments:

- *NLS* – a SUNNonlinSol object.

Return value: The SUNNonlinSol type identifier (of type `int`) will be one of the following:

- `SUNNONLINEARSOLVER_ROOTFIND` – `0`, the SUNNonlinSol module solves $F(y) = 0$.
- `SUNNONLINEARSOLVER_FIXEDPOINT` – `1`, the SUNNonlinSol module solves $G(y) = y$.

int **SUNNonlinSolInitialize**(*SUNNonlinearSolver* NLS)
This *optional* function handles nonlinear solver initialization and may perform any necessary memory allocations.

Arguments:

- *NLS* – a SUNNonlinSol object.

Return value: The return value is zero for a successful call and a negative value for a failure.

Notes: It is assumed all solver-specific options have been set prior to calling *SUNNonlinSolInitialize()*. SUNNonlinSol implementations that do not require initialization may set this operation to `NULL`.

int **SUNNonlinSolSetup**(*SUNNonlinearSolver* NLS, *N_Vector* y, void *mem)
This *optional* function performs any solver setup needed for a nonlinear solve.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *y* – the initial guess passed to the nonlinear solver.
- *mem* – the SUNDIALS integrator memory structure.

Return value: The return value is zero for a successful call and a negative value for a failure.

Notes: SUNDIALS integrators call `SUNonlinSolSetup()` before each step attempt. SUNNonlinSol implementations that do not require setup may set this operation to `NULL`.

int **SUNNonlinSolSolve**(*SUNNonlinearSolver* NLS, *N_Vector* y0, *N_Vector* ycor, *N_Vector* w, *realtype* tol, *booleantype* callLSetup, void *mem)
This *required* function solves the nonlinear system $F(y) = 0$ or $G(y) = y$.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *y0* – the predicted value for the new solution state. This *must* remain unchanged throughout the solution process.
- *ycor* – on input the initial guess for the correction to the predicted state (zero) and on output the final correction to the predicted state.
- *w* – the solution error weight vector used for computing weighted error norms.
- *tol* – the requested solution tolerance in the weighted root-mean-squared norm.

- *callLSetup* – a flag indicating that the integrator recommends for the linear solver setup function to be called.
- *mem* – the SUNDIALS integrator memory structure.

Return value: The return value is zero for a successul solve, a positive value for a recoverable error (i.e., the solve failed and the integrator should reduce the step size and reattempt the step), and a negative value for an unrecoverable error (i.e., the solve failed the and the integrator should halt and return an error to the user).

int **SUNNonlinSolFree**(*SUNNonlinearSolver* NLS)

This *optional* function frees any memory allocated by the nonlinear solver.

Arguments:

- *NLS* – a SUNNonlinSol object.

Return value: The return value should be zero for a successful call, and a negative value for a failure. SUNNonlinSol implementations that do not allocate data may set this operation to `NULL`.

9.1.2 SUNNonlinearSolver "set" functions

The following functions are used to supply nonlinear solver modules with functions defined by the SUNDIALS integrators and to modify solver parameters. Only the routine for setting the nonlinear system defining function (*SUNNonlinSolSetSysFn()*) is required. All other set functions are optional.

int **SUNNonlinSolSetSysFn**(*SUNNonlinearSolver* NLS, *SUNNonlinSolSysFn* SysFn)

This *required* function is used to provide the nonlinear solver with the function defining the nonlinear system. This is the function $F(y)$ in $F(y) = 0$ for `SUNNONLINEARSOLVER_ROOTFIND` modules or $G(y)$ in $G(y) = y$ for `SUNNONLINEARSOLVER_FIXEDPOINT` modules.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *SysFn* – the function defining the nonlinear system. See §9.1.4 for the definition of *SUNNonlinSolSysFn*.

Return value: The return value should be zero for a successful call, and a negative value for a failure.

int **SUNNonlinSolSetLSetupFn**(*SUNNonlinearSolver* NLS, *SUNNonlinSolLSetupFn* SetupFn)

This *optional* function is called by SUNDIALS integrators to provide the nonlinear solver with access to its linear solver setup function.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *SetupFn* – a wrapper function to the SUNDIALS integrator's linear solver setup function. See §9.1.4 for the definition of *SUNNonlinSolLSetupFn*.

Return value: The return value should be zero for a successful call, and a negative value for a failure.

Notes: The *SUNNonlinSolLSetupFn* function sets up the linear system $Ax = b$ where $A = \frac{\partial F}{\partial y}$ is the linearization of the nonlinear residual function $F(y) = 0$ (when using SUNLinSol direct linear solvers) or calls the user-defined preconditioner setup function (when using SUNLinSol iterative linear solvers). SUNNonlinSol implementations that do not require solving this system, do not utilize SUNLinSol linear solvers, or use SUNLinSol linear solvers that do not require setup may set this operation to `NULL`.

int **SUNNonlinSolSetLSolveFn**(*SUNNonlinearSolver* NLS, *SUNNonlinSolLSolveFn* SolveFn)

This *optional* function is called by SUNDIALS integrators to provide the nonlinear solver with access to its linear solver solve function.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *SolveFn* – a wrapper function to the SUNDIALS integrator's linear solver solve function. See §9.1.4 for the definition of `SUNNonlinSolLSolveFn`.

Return value: The return value should be zero for a successful call, and a negative value for a failure.

Notes: The `SUNNonlinSolLSolveFn` function solves the linear system $Ax = b$ where $A = \frac{\partial F}{\partial y}$ is the linearization of the nonlinear residual function $F(y) = 0$. SUNNonlinSol implementations that do not require solving this system or do not use SUNLinSol linear solvers may set this operation to `NULL`.

int **SUNNonlinSolSetConvTestFn**(*SUNNonlinearSolver* NLS, *SUNNonlinSolConvTestFn* CTestFn, void *ctest_data)

This *optional* function is used to provide the nonlinear solver with a function for determining if the nonlinear solver iteration has converged. This is typically called by SUNDIALS integrators to define their nonlinear convergence criteria, but may be replaced by the user.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *CTestFn* – a SUNDIALS integrator's nonlinear solver convergence test function. See §9.1.4 for the definition of `SUNNonlinSolConvTestFn`.
- *ctest_data* – is a data pointer passed to *CTestFn* every time it is called.

Return value: The return value should be zero for a successful call, and a negative value for a failure.

Notes: SUNNonlinSol implementations utilizing their own convergence test criteria may set this function to `NULL`.

int **SUNNonlinSolSetMaxIters**(*SUNNonlinearSolver* NLS, int maxiters)

This *optional* function sets the maximum number of nonlinear solver iterations. This is typically called by SUNDIALS integrators to define their default iteration limit, but may be adjusted by the user.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *maxiters* – the maximum number of nonlinear iterations.

Return value: The return value should be zero for a successful call, and a negative value for a failure (e.g., $maxiters < 1$).

9.1.3 SUNNonlinearSolver "get" functions

The following functions allow SUNDIALS integrators to retrieve nonlinear solver statistics. The routines to get the number of iterations in the most recent solve (`SUNNonlinSolGetNumIters()`) and number of convergence failures are optional. The routine to get the current nonlinear solver iteration (`SUNNonlinSolGetCurIter()`) is required when using the convergence test provided by the SUNDIALS integrator or when using an iterative SUNLinSol linear solver module; otherwise `SUNNonlinSolGetCurIter()` is optional.

int **SUNNonlinSolGetNumIters**(*SUNNonlinearSolver* NLS, long int *niters)

This *optional* function returns the number of nonlinear solver iterations in the most recent solve. This is typically called by the SUNDIALS integrator to store the nonlinear solver statistics, but may also be called by the user.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *niters* – the total number of nonlinear solver iterations.

Return value: The return value should be zero for a successful call, and a negative value for a failure.

int **SUNNonlinSolGetCurIter**(*SUNNonlinearSolver* NLS, int *iter)

This function returns the iteration index of the current nonlinear solve. This function is *required* when using SUNDIALS integrator-provided convergence tests or when using an iterative SUNLinSol linear solver module; otherwise it is *optional*.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *iter* – the nonlinear solver iteration in the current solve starting from zero.

Return value: The return value should be zero for a successful call, and a negative value for a failure.

int **SUNNonlinSolGetNumConvFails**(*SUNNonlinearSolver* NLS, long int *nconvfails)

This *optional* function returns the number of nonlinear solver convergence failures in the most recent solve. This is typically called by the SUNDIALS integrator to store the nonlinear solver statistics, but may also be called by the user.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *nconvfails* – the total number of nonlinear solver convergence failures.

Return value: The return value should be zero for a successful call, and a negative value for a failure.

9.1.4 Functions provided by SUNDIALS integrators

To interface with SUNNonlinSol modules, the SUNDIALS integrators supply a variety of routines for evaluating the nonlinear system, calling the SUNLinSol setup and solve functions, and testing the nonlinear iteration for convergence. These integrator-provided routines translate between the user-supplied ODE or DAE systems and the generic interfaces to the nonlinear or linear systems of equations that result in their solution. The functions provided to a SUNNonlinSol module have types defined in the header file `sundials/sundials_nonlinearsolver.h`; these are also described below.

typedef int (***SUNNonlinSolSysFn**)(*N_Vector* ycor, *N_Vector* F, void *mem)

These functions evaluate the nonlinear system $F(y)$ for `SUNNONLINEARSOLVER_ROOTFIND` type modules or $G(y)$ for `SUNNONLINEARSOLVER_FIXEDPOINT` type modules. Memory for F must by be allocated prior to calling this function. The vector *ycor* will be left unchanged.

Arguments:

- *ycor* – is the current correction to the predicted state at which the nonlinear system should be evaluated.
- *F* – is the output vector containing $F(y)$ or $G(y)$, depending on the solver type.
- *mem* – is the SUNDIALS integrator memory structure.

Return value: The return value is zero for a successul solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.

Notes: SUNDIALS integrators formulate nonlinear systems as a function of the correction to the predicted solution. On each call to the nonlinear system function the integrator will compute and store the current solution based on the input correction. Additionally, the residual will store the value of the ODE right-hand side function or DAE residual used in computing the nonlinear system. These stored values are then directly used in the integrator-supplied linear solver setup and solve functions as applicable.

typedef int (***SUNNonlinSolLSetupFn**)(*booleantype* jbad, *booleantype* *jcur, void *mem)

These functions are wrappers to the SUNDIALS integrator's function for setting up linear solves with SUNLinSol modules.

Arguments:

- *jbad* – is an input indicating whether the nonlinear solver believes that A has gone stale (`SUNTRUE`) or not (`SUNFALSE`).
- *jcur* – is an output indicating whether the routine has updated the Jacobian A (`SUNTRUE`) or not (`SUNFALSE`).
- *mem* – is the SUNDIALS integrator memory structure.

Return value: The return value is zero for a successul solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.

Notes: The *SUNNonlinSolLSetupFn* function sets up the linear system $Ax = b$ where $A = \frac{\partial F}{\partial y}$ is the linearization of the nonlinear residual function $F(y) = 0$ (when using SUNLinSol direct linear solvers) or calls the user-defined preconditioner setup function (when using SUNLinSol iterative linear solvers). SUNNonlinSol implementations that do not require solving this system, do not utilize SUNLinSol linear solvers, or use SUNLinSol linear solvers that do not require setup may ignore these functions.

As discussed in the description of *SUNNonlinSolSysFn*, the linear solver setup function assumes that the nonlinear system function has been called prior to the linear solver setup function as the setup will utilize saved values from the nonlinear system evaluation (e.g., the updated solution).

typedef int (***SUNNonlinSolLSolveFn**)(*N_Vector* b, void *mem)

These functions are wrappers to the SUNDIALS integrator's function for solving linear systems with SUNLinSol modules.

Arguments:

- *b* – contains the right-hand side vector for the linear solve on input and the solution to the linear system on output.
- *mem* – is the SUNDIALS integrator memory structure.

Return value: The return value is zero for a successul solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.

Notes: The *SUNNonlinSolLSolveFn* function solves the linear system $Ax = b$ where $A = \frac{\partial F}{\partial y}$ is the linearization of the nonlinear residual function $F(y) = 0$. SUNNonlinSol implementations that do not require solving this system or do not use SUNLinSol linear solvers may ignore these functions.

As discussed in the description of *SUNNonlinSolSysFn*, the linear solver solve function assumes that the nonlinear system function has been called prior to the linear solver solve function as the setup may utilize saved values from the nonlinear system evaluation (e.g., the updated solution).

typedef int (***SUNNonlinSolConvTestFn**)(*SUNNonlinearSolver* NLS, *N_Vector* ycor, *N_Vector* del, *realtype* tol, *N_Vector* ewt, void *ctest_data)

These functions are SUNDIALS integrator-specific convergence tests for nonlinear solvers and are typically supplied by each SUNDIALS integrator, but users may supply custom problem-specific versions as desired.

Arguments:

- *NLS* – is the SUNNonlinSol object.
- *ycor* – is the current correction (nonlinear iterate).
- *del* – is the difference between the current and prior nonlinear iterates.
- *tol* – is the nonlinear solver tolerance.
- *ewt* – is the weight vector used in computing weighted norms.
- *ctest_data* – is the data pointer provided to *SUNNonlinSolSetConvTestFn()*.

Return value: The return value of this routine will be a negative value if an unrecoverable error occurred or one of the following:

- SUN_NLS_SUCCESS – the iteration is converged.
- SUN_NLS_CONTINUE – the iteration has not converged, keep iterating.
- SUN_NLS_CONV_RECVR – the iteration appears to be diverging, try to recover.

Notes: The tolerance passed to this routine by SUNDIALS integrators is the tolerance in a weighted root-mean-squared norm with error weight vector ewt. SUNNonlinSol modules utilizing their own convergence criteria may ignore these functions.

9.1.5 SUNNonlinearSolver return codes

The functions provided to SUNNonlinSol modules by each SUNDIALS integrator, and functions within the SUNDIALS-provided SUNNonlinSol implementations, utilize a common set of return codes shown in Table 9.1. Here, negative values correspond to non-recoverable failures, positive values to recoverable failures, and zero to a successful call.

Table 9.1: Description of the SUNNonlinearSolver return codes.

Name	Value	Description
SUN_NLS_SUCCESS	0	successful call or converged solve
SUN_NLS_CONTINUE	901	the nonlinear solver is not converged, keep iterating
SUN_NLS_CONV_RECVR	902	the nonlinear solver appears to be diverging, try to recover
SUN_NLS_MEM_NULL	-901	a memory argument is NULL
SUN_NLS_MEM_FAIL	-902	a memory access or allocation failed
SUN_NLS_ILL_INPUT	-903	an illegal input option was provided
SUN_NLS_VECTOROP_ERR	-904	a NVECTOR operation failed
SUN_NLS_EXT_FAIL	-905	an external library call returned an error

9.1.6 The generic SUNNonlinearSolver module

SUNDIALS integrators interact with specific SUNNonlinSol implementations through the generic SUNNonlinSol module on which all other SUNNonlinSol implementations are built. The SUNNonlinearSolver type is a pointer to a structure containing an implementation-dependent *content* field and an *ops* field. The type SUNNonlinearSolver is defined as follows:

typedef struct _generic_SUNNonlinearSolver ***SUNNonlinearSolver**

and the generic structure is defined as

```
struct _generic_SUNNonlinearSolver {
  void *content;
  struct _generic_SUNNonlinearSolver_Ops *ops;
};
```

where the _generic_SUNNonlinearSolver_Ops structure is a list of pointers to the various actual nonlinear solver operations provided by a specific implementation. The _generic_SUNNonlinearSolver_Ops structure is defined as

```
struct _generic_SUNNonlinearSolver_Ops {
  SUNNonlinearSolver_Type (*gettype)(SUNNonlinearSolver);
  int                     (*initialize)(SUNNonlinearSolver);
```

(continues on next page)

(continued from previous page)

```
  int                    (*setup)(SUNNonlinearSolver, N_Vector, void*);
  int                    (*solve)(SUNNonlinearSolver, N_Vector, N_Vector,
                                  N_Vector, realtype, booleantype, void*);
  int                    (*free)(SUNNonlinearSolver);
  int                    (*setsysfn)(SUNNonlinearSolver, SUNNonlinSolSysFn);
  int                    (*setlsetupfn)(SUNNonlinearSolver, SUNNonlinSolLSetupFn);
  int                    (*setlsolvefn)(SUNNonlinearSolver, SUNNonlinSolLSolveFn);
  int                    (*setctestfn)(SUNNonlinearSolver, SUNNonlinSolConvTestFn,
                                       void*);
  int                    (*setmaxiters)(SUNNonlinearSolver, int);
  int                    (*getnumiters)(SUNNonlinearSolver, long int*);
  int                    (*getcuriter)(SUNNonlinearSolver, int*);
  int                    (*getnumconvfails)(SUNNonlinearSolver, long int*);
};
```

The generic SUNNonlinSol module defines and implements the nonlinear solver operations defined in §9.1.1–§9.1.3. These routines are in fact only wrappers to the nonlinear solver operations provided by a particular SUNNonlinSol implementation, which are accessed through the ops field of the `SUNNonlinearSolver` structure. To illustrate this point we show below the implementation of a typical nonlinear solver operation from the generic SUNNonlinSol module, namely *SUNNonlinSolSolve()*, which solves the nonlinear system and returns a flag denoting a successful or failed solve:

```
int SUNNonlinSolSolve(SUNNonlinearSolver NLS,
                      N_Vector y0, N_Vector y,
                      N_Vector w, realtype tol,
                      booleantype callLSetup, void* mem)
{
  return((int) NLS->ops->solve(NLS, y0, y, w, tol, callLSetup, mem));
}
```

9.1.7 Implementing a Custom SUNNonlinearSolver Module

A SUNNonlinSol implementation *must* do the following:

- Specify the content of the SUNNonlinSol module.
- Define and implement the required nonlinear solver operations defined in §9.1.1–§9.1.3. Note that the names of the module routines should be unique to that implementation in order to permit using more than one SUNNonlinSol module (each with different `SUNNonlinearSolver` internal data representations) in the same code.
- Define and implement a user-callable constructor to create a `SUNNonlinearSolver` object.

To aid in the creation of custom `SUNNonlinearSolver` modules, the generic `SUNNonlinearSolver` module provides the utility functions *SUNNonlinSolNewEmpty()* and `SUNNonlinsolFreeEmpty()`. When used in custom `SUNNonlinearSolver` constructors these functions will ease the introduction of any new optional nonlinear solver operations to the `SUNNonlinearSolver` API by ensuring that only required operations need to be set.

SUNNonlinearSolver **SUNNonlinSolNewEmpty()**

This function allocates a new generic `SUNNonlinearSolver` object and initializes its content pointer and the function pointers in the operations structure to `NULL`.

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

void **SUNNonlinSolFreeEmpty**(*SUNNonlinearSolver* NLS)

This routine frees the generic `SUNNonlinearSolver` 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:

- *NLS* – a SUNNonlinearSolver object

Additionally, a `SUNNonlinearSolver` implementation *may* do the following:

- Define and implement additional user-callable "set" routines acting on the `SUNNonlinearSolver` object, e.g., for setting various configuration options to tune the performance of the nonlinear solve algorithm.
- Provide additional user-callable "get" routines acting on the `SUNNonlinearSolver` object, e.g., for returning various solve statistics.

9.2 IDAS SUNNonlinearSolver interface

As discussed in Chapter §2 each integration step requires the (approximate) solution of the nonlinear system

$$G(y_n) = F\left(t_n, y_n, h_n^{-1} \sum_{i=0}^{q} \alpha_{n,i} y_{n-i}\right) = 0.$$

Rather than solving this system for the new state y_n IDAS reformulates the system to solve for the correction y_{cor} to the predicted new state y_{pred} and its derivative $\dot{y}_{pred}$ so that $y_n = y_{pred} + y_{cor}$ and $\dot{y}_n = \dot{y}_{pred} + h_n^{-1}\, \alpha_{n,0}\, y_{cor}$. The nonlinear system rewritten in terms of y_{cor} is

$$G(y_{cor}) = F\left(t_n,\, y_{pred} + y_{cor},\, \dot{y}_{pred} + \alpha y_{cor}\right) = 0. \tag{9.1}$$

where $\alpha = h_n^{-1}\, \alpha_{n,0}$.

Similarly in the forward sensitivity analysis case the nonlinear system is also reformulated in terms of the correction to the predicted sensitivities.

The nonlinear system function provided by IDAS to the nonlinear solver module internally updates the current value of the new state and its derivative based on the current corretion passed to the function (as well as the sensitivities). These values are used when calling the DAE residual function and when setting up linear solves (e.g., for updating the Jacobian or preconditioner).

IDAS provides several advanced functions that will not be needed by most users, but might be useful for users who choose to provide their own implementation of the `SUNNonlinearSolver` API. For example, such a user might need access to the current y and $\dot{y}$ vectors to compute Jacobian data.

int **IDAGetCurrentCj**(void *ida_mem, *realtype* *cj)

The function *IDAGetCurrentCj()* returns the scalar c_j which is proportional to the inverse of the step size (α in (2.6)).

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `cj` – the value of c_j.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The IDAS memory block is `NULL`.

int **IDAGetCurrentY**(void *ida_mem, *N_Vector* *ycur)
The function *IDAGetCurrentY()* returns the current y vector.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `y` – the current y vector.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The IDAS memory block is `NULL`.

int **IDAGetCurrentYp**(void *ida_mem, *N_Vector* *ypcur)
The function *IDAGetCurrentYp()* returns the current $\dot{y}$ vector.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `yp` – the current $\dot{y}$ vector.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The IDAS memory block is `NULL`.

int **IDAGetCurrentYSens**(void *ida_mem, *N_Vector* **yyS)
The function *IDAGetCurrentYSens()* returns the current sensitivity vector array.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `yyS` – pointer to the vector array that is set to the array of sensitivity vectors.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.

int **IDAGetCurrentYpSens**(void *ida_mem, *N_Vector* **ypS)
The function *IDAGetCurrentYpSens()* returns the derivative the current sensitivity vector array.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `ypS` – pointer to the vector array that is set to the array of sensitivity vector derivatives.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.

int **IDAGetNonlinearSystemData**(void *ida_mem, *realtype* *tcur, *N_Vector* *yypred, *N_Vector* *yppred, *N_Vector* *yyn, *N_Vector* *ypn, *N_Vector* *res, *realtype* *cj, void **user_data)
The function *IDAGetNonlinearSystemData()* returns all internal data required to construct the current nonlinear system (9.1).

Arguments:

- `ida_mem` – pointer to the IDAS memory block.

- `tcur` – current value of the independent variable t_n.
- `yypred` – predicted value of y_{pred} at t_n.
- `yppred` – predicted value of $\dot{y}_{pred}$ at t_n.
- `yyn` – the vector y_n. This vector may not be current and may need to be filled (see the note below).
- `ypn` – the vector $\dot{y}_n$. This vector may not be current and may need to be filled (see the note below).
- `res` – the resiudal function evaluated at the current time and state, $F(t_n, y_n, \dot{y}_n)$. This vector may not be current and may need to be filled (see the note below).
- `cj` – the scalar c_j which is proportional to the inverse of the step size (α in (9.1)).
- `user_data` – pointer to the user-defined data structures.

Return value:

- `IDA_SUCCESS` – The optional output values have been successfully set.
- `IDA_MEM_NULL` – The IDAS memory block is `NULL`.

Notes: This routine is intended for users who wish to attach a custom *SUNNonlinSolSysFn* to an existing `SUNNonlinearSolver` object (through a call to *SUNNonlinSolSetSysFn()*) or who need access to nonlinear system data to compute the nonlinear system fucntion as part of a custom `SUNNonlinearSolver` object.

When supplying a custom *SUNNonlinSolSysFn* to an existing `SUNNonlinearSolver` object, the user should call *IDAGetNonlinearSystemData()* inside the nonlinear system function to access the requisite data for evaluting the nonlinear system function of their choosing. Additionlly, if the `SUNNonlinearSolver` object (existing or custom) leverages the *SUNNonlinSolLSetupFn* and/or *SUNNonlinSolLSolveFn* functions supplied by IDAS (through calls to *SUNNonlinSolSetLSetupFn()* and *SUNNonlinSolSetLSolveFn()* respectively) the vectors `yyn` and `ypn`, and `res` must be filled in by the user's *SUNNonlinSolSysFn* with the current state and corresponding evaluation of the right-hand side function respectively i.e.,

$$\begin{aligned} yyn &= y_{pred} + y_{cor}, \\ ypn &= \dot{y}_{pred} + \alpha \dot{y}_{cor}, \\ res &= F\left(t_n, y_n, \dot{y}_n\right), \end{aligned}$$

where y_{cor} was the first argument supplied to the *SUNNonlinSolSysFn*. If this function is called as part of a custom linear solver (i.e., the default *SUNNonlinSolSysFn* is used) then the vectors `yn`, `ypn` and `res` are only current when *IDAGetNonlinearSystemData()* is called after an evaluation of the nonlinear system function.

int **IDAGetNonlinearSystemDataSens**(void *ida_mem, *realtype* *tcur, *N_Vector* **yySpred, *N_Vector* **ypSpred, *N_Vector* **yySn, *N_Vector* **ypSn, *realtype* *cj, void **user_data)

The function *IDAGetNonlinearSystemDataSens()* returns all internal sensitivity data required to construct the current nonlinear system (9.1).

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `tcur` – current value of the independent variable t_n.
- `yySpred` – predicted value of $yS_{i,pred}$ at t_n for $i = 0 \ldots N_s - 1$.
- `ypSpred` – predicted value of $\dot{y}S_{i,pred}$ at t_n for $i = 0 \ldots N_s - 1$.
- `yySn` – the vectors $yS_{i,n}$. These vectors may be not current see the note below.
- `ypSn` – the vectors $\dot{y}S_{i,n}$. These vectors may be not current see the note below.

- `cj` – the scalar c_j which is proportional to the inverse of the step size α in (2.6).
- `user_data` – pointer to the user-defined data structures

Return value:

- `IDA_SUCCESS` – The optional output values have been successfully set.
- `IDA_MEM_NULL` – The `ida_mem` pointer is `NULL`.

Notes: This routine is intended for users who wish to attach a custom *SUNNonlinSolSysFn* to an existing `SUNNonlinearSolver` object (through a call to *SUNNonlinSolSetSysFn()*) or who need access to nonlinear system data to compute the nonlinear system fucntion as part of a custom `SUNNonlinearSolver` object. When supplying a custom *SUNNonlinSolSysFn* to an existing `SUNNonlinearSolver` object, the user should call *IDAGetNonlinearSystemDataSens()* inside the nonlinear system function to access the requisite data for evaluting the nonlinear system function of their choosing. Additionlly, if the the vectors `yySn` and `ypSn` are provided as additional workspace and do not need to be filled in by the user's *SUNNonlinSolSysFn*. If this function is called as part of a custom linear solver (i.e., the default *SUNNonlinSolSysFn* is used) then the vectors `yySn` and `ypSn` are only current when *IDAGetNonlinearSystemDataSens()* is called after an evaluation of the nonlinear system function.

int **IDAComputeY**(void *ida_mem, *N_Vector* ycor, *N_Vector* y)

The function computes the current $y(t)$ vector based on the given correction vector from the nonlinear solver.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `ycor` – the correction.
- `y` – the output vector.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The IDAS memory block is `NULL`.

int **IDAComputeYp**(void *ida_mem, *N_Vector* ycor, *N_Vector* yp)

The function computes $\dot{y}(t)$ based on the given correction vector from the nonlinear solver.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `ycor` – the correction.
- `yp` – the output vector array.

Return value:

- `IDA_SUCCESS` – The optional output value has been successfully set.
- `IDA_MEM_NULL` – The IDAS memory block is `NULL`.

int **IDAComputeYSens**(void *ida_mem, *N_Vector* *ycorS, *N_Vector* *yys)

The function computes the sensitivities based on the given correction vector from the nonlinear solver.

Arguments:

- `ida_mem` – pointer to the IDAS memory block.
- `ycorS` – the correction.
- `yyS` – the output vector array.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

int **IDAComputeYpSens**(void *ida_mem, *N_Vector* *ycorS, *N_Vector* *ypS)

The function computes the sensitivity derivatives based on the given correction vector from the nonlinear solver.

Arguments:

- ida_mem – pointer to the IDAS memory block.
- ycorS – the correction.
- ypS – the output vector array.

Return value:

- IDA_SUCCESS – The optional output value has been successfully set.
- IDA_MEM_NULL – The ida_mem pointer is NULL.

9.3 The SUNNonlinSol_Newton implementation

This section describes the SUNNonlinSol implementation of Newton's method. To access the SUNNonlinSol_Newton module, include the header file sunnonlinsol/sunnonlinsol_newton.h. We note that the SUNNonlinSol_Newton module is accessible from SUNDIALS integrators *without* separately linking to the libsundials_sunnonlinsol-newton module library.

9.3.1 SUNNonlinSol_Newton description

To find the solution to

$$F(y) = 0 \tag{9.2}$$

given an initial guess $y^{(0)}$, Newton's method computes a series of approximate solutions

$$y^{(m+1)} = y^{(m)} + \delta^{(m+1)}$$

where m is the Newton iteration index, and the Newton update $\delta^{(m+1)}$ is the solution of the linear system

$$A(y^{(m)})\delta^{(m+1)} = -F(y^{(m)})\,, \tag{9.3}$$

in which A is the Jacobian matrix

$$A \equiv \partial F/\partial y\,. \tag{9.4}$$

Depending on the linear solver used, the SUNNonlinSol_Newton module will employ either a Modified Newton method or an Inexact Newton method [5, 10, 19, 21, 36]. When used with a direct linear solver, the Jacobian matrix A is held constant during the Newton iteration, resulting in a Modified Newton method. With a matrix-free iterative linear solver, the iteration is an Inexact Newton method.

In both cases, calls to the integrator-supplied *SUNNonlinSolLSetupFn* function are made infrequently to amortize the increased cost of matrix operations (updating A and its factorization within direct linear solvers, or updating the preconditioner within iterative linear solvers). Specifically, SUNNonlinSol_Newton will call the *SUNNonlinSolLSetupFn* function in two instances:

(a) when requested by the integrator (the input callLSetSetup is SUNTRUE) before attempting the Newton iteration, or

(b) when reattempting the nonlinear solve after a recoverable failure occurs in the Newton iteration with stale Jacobian information (`jcur` is `SUNFALSE`). In this case, SUNNonlinSol_Newton will set `jbad` to `SUNTRUE` before calling the *SUNNonlinSolLSetupFn()* function.

Whether the Jacobian matrix A is fully or partially updated depends on logic unique to each integrator-supplied `SUNNonlinSolSetupFn` routine. We refer to the discussion of nonlinear solver strategies provided in the package-specific Mathematics section of the documentation for details.

The default maximum number of iterations and the stopping criteria for the Newton iteration are supplied by the SUNDIALS integrator when SUNNonlinSol_Newton is attached to it. Both the maximum number of iterations and the convergence test function may be modified by the user by calling the *SUNNonlinSolSetMaxIters()* and/or *SUNNonlinSolSetConvTestFn()* functions after attaching the SUNNonlinSol_Newton object to the integrator.

9.3.2 SUNNonlinSol_Newton functions

The SUNNonlinSol_Newton module provides the following constructor for creating the `SUNNonlinearSolver` object.

SUNNonlinearSolver **SUNNonlinSol_Newton**(*N_Vector* y, *SUNContext* sunctx)

This creates a `SUNNonlinearSolver` object for use with SUNDIALS integrators to solve nonlinear systems of the form $F(y) = 0$ using Newton's method.

Arguments:

- *y* – a template for cloning vectors needed within the solver.
- *sunctx* – the *SUNContext* object (see §4.2)

Return value: A SUNNonlinSol object if the constructor exits successfully, otherwise it will be `NULL`.

The SUNNonlinSol_Newton module implements all of the functions defined in §9.1.1–§9.1.3 except for *SUNNonlinSolSetup()*. The SUNNonlinSol_Newton functions have the same names as those defined by the generic SUNNonlinSol API with `_Newton` appended to the function name. Unless using the SUNNonlinSol_Newton module as a standalone nonlinear solver the generic functions defined in §9.1.1–§9.1.3 should be called in favor of the SUNNonlinSol_Newton-specific implementations.

The SUNNonlinSol_Newton module also defines the following user-callable function.

int **SUNNonlinSolGetSysFn_Newton**(*SUNNonlinearSolver* NLS, *SUNNonlinSolSysFn* *SysFn)

This returns the residual function that defines the nonlinear system.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *SysFn* – the function defining the nonlinear system.

Return value: The return value should be zero for a successful call, and a negative value for a failure.

Notes: This function is intended for users that wish to evaluate the nonlinear residual in a custom convergence test function for the SUNNonlinSol_Newton module. We note that SUNNonlinSol_Newton will not leverage the results from any user calls to *SysFn*.

int **SUNNonlinSolSetInfoFile_Newton**(*SUNNonlinearSolver* NLS, FILE *info_file)

This sets the output file where all informative (non-error) messages should be directed.

Arguments:

- *NLS* – a SUNNonlinSol object.
- ***info_file* – pointer to output file (`stdout` by default);** a `NULL` input will disable output.

Return value:

- SUN_NLS_SUCCESS if successful.
- SUN_NLS_MEM_NULL if the SUNNonlinSol memory was NULL.
- SUN_NLS_ILL_INPUT if SUNDIALS was not built with monitoring enabled.

Notes: This function is intended for users that wish to monitor the nonlinear solver progress. By default, the file pointer is set to stdout.

Warning: SUNDIALS must be built with the CMake option SUNDIALS_BUILD_WITH_MONITORING to utilize this function. See §11.1.2 for more information.

int **SUNNonlinSolSetPrintLevel_Newton**(*SUNNonlinearSolver* NLS, int print_level)
This specifies the level of verbosity of the output.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *print_level* – flag indicating level of verbosity; must be one of:
  - 0, no information is printed (default).
  - 1, for each nonlinear iteration the residual norm is printed.

Return value:

- SUN_NLS_SUCCESS if successful.
- SUN_NLS_MEM_NULL if the SUNNonlinearSolver memory was NULL.
- SUN_NLS_ILL_INPUT if SUNDIALS was not built with monitoring enabled, or the print level value was invalid.

Notes: This function is intended for users that wish to monitor the nonlinear solver progress. By default, the print level is 0.

Warning: SUNDIALS must be built with the CMake option SUNDIALS_BUILD_WITH_MONITORING to utilize this function. See §11.1.2 for more information.

9.3.3 SUNNonlinSol_Newton content

The *content* field of the SUNNonlinSol_Newton module is the following structure.

```
struct _SUNNonlinearSolverContent_Newton {

  SUNNonlinSolSysFn      Sys;
  SUNNonlinSolLSetupFn   LSetup;
  SUNNonlinSolLSolveFn   LSolve;
  SUNNonlinSolConvTestFn CTest;

  N_Vector    delta;
  booleantype jcur;
  int         curiter;
  int         maxiters;
  long int    niters;
```

(continues on next page)

(continued from previous page)

```
  long int    nconvfails;
  void*       ctest_data;

  int         print_level;
  FILE*       info_file;
};
```

These entries of the *content* field contain the following information:

- `Sys` – the function for evaluating the nonlinear system,
- `LSetup` – the package-supplied function for setting up the linear solver,
- `LSolve` – the package-supplied function for performing a linear solve,
- `CTest` – the function for checking convergence of the Newton iteration,
- `delta` – the Newton iteration update vector,
- `jcur` – the Jacobian status (`SUNTRUE` = current, `SUNFALSE` = stale),
- `curiter` – the current number of iterations in the solve attempt,
- `maxiters` – the maximum number of Newton iterations allowed in a solve,
- `niters` – the total number of nonlinear iterations across all solves,
- `nconvfails` – the total number of nonlinear convergence failures across all solves,
- `ctest_data` – the data pointer passed to the convergence test function,
- `print_level` - controls the amount of information to be printed to the info file,
- `info_file` - the file where all informative (non-error) messages will be directed.

9.4 The SUNNonlinSol_FixedPoint implementation

This section describes the SUNNonlinSol implementation of a fixed point (functional) iteration with optional Anderson acceleration. To access the SUNNonlinSol_FixedPoint module, include the header file `sunnonlinsol/sunnonlinsol_fixedpoint.h`. We note that the SUNNonlinSol_FixedPoint module is accessible from SUNDIALS integrators *without* separately linking to the `libsundials_sunnonlinsolfixedpoint` module library.

9.4.1 SUNNonlinSol_FixedPoint description

To find the solution to

$$G(y) = y \tag{9.5}$$

given an initial guess $y^{(0)}$, the fixed point iteration computes a series of approximate solutions

$$y^{(n+1)} = G(y^{(n)}) \tag{9.6}$$

where n is the iteration index. The convergence of this iteration may be accelerated using Anderson's method [1, 23, 41, 48]. With Anderson acceleration using subspace size m, the series of approximate solutions can be formulated as the linear combination

$$y^{(n+1)} = \beta \sum_{i=0}^{m_n} \alpha_i^{(n)} G(y^{(n-m_n+i)}) + (1-\beta) \sum_{i=0}^{m_n} \alpha_i^{(n)} y_{n-m_n+i} \tag{9.7}$$

where $m_n = \min\{m, n\}$ and the factors

$$\alpha^{(n)} = (\alpha_0^{(n)}, \ldots, \alpha_{m_n}^{(n)})$$

solve the minimization problem $\min_{\alpha} \|F_n \alpha^T\|_2$ under the constraint that $\sum_{i=0}^{m_n} \alpha_i = 1$ where

$$F_n = (f_{n-m_n}, \ldots, f_n)$$

with $f_i = G(y^{(i)}) - y^{(i)}$. Due to this constraint, in the limit of $m = 0$ the accelerated fixed point iteration formula (9.7) simplifies to the standard fixed point iteration (9.6).

Following the recommendations made in [48], the SUNNonlinSol_FixedPoint implementation computes the series of approximate solutions as

$$y^{(n+1)} = G(y^{(n)}) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta g_{n-m_n+i} - (1-\beta)(f(y^{(n)}) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta f_{n-m_n+i}) \tag{9.8}$$

with $\Delta g_i = G(y^{(i+1)}) - G(y^{(i)})$ and where the factors

$$\gamma^{(n)} = (\gamma_0^{(n)}, \ldots, \gamma_{m_n-1}^{(n)})$$

solve the unconstrained minimization problem $\min_{\gamma} \|f_n - \Delta F_n \gamma^T\|_2$ where

$$\Delta F_n = (\Delta f_{n-m_n}, \ldots, \Delta f_{n-1}),$$

with $\Delta f_i = f_{i+1} - f_i$. The least-squares problem is solved by applying a QR factorization to $\Delta F_n = Q_n R_n$ and solving $R_n \gamma = Q_n^T f_n$.

The acceleration subspace size m is required when constructing the SUNNonlinSol_FixedPoint object. The default maximum number of iterations and the stopping criteria for the fixed point iteration are supplied by the SUNDIALS integrator when SUNNonlinSol_FixedPoint is attached to it. Both the maximum number of iterations and the convergence test function may be modified by the user by calling *SUNNonlinSolSetMaxIters()* and *SUNNonlinSolSetConvTestFn()* after attaching the SUNNonlinSol_FixedPoint object to the integrator.

9.4.2 SUNNonlinSol_FixedPoint functions

The SUNNonlinSol_FixedPoint module provides the following constructor for creating the `SUNNonlinearSolver` object.

SUNNonlinearSolver **SUNNonlinSol_FixedPoint**(*N_Vector* y, int m, *SUNContext* sunctx)

This creates a `SUNNonlinearSolver` object for use with SUNDIALS integrators to solve nonlinear systems of the form $G(y) = y$.

Arguments:

- *y* – a template for cloning vectors needed within the solver.
- *m* – the number of acceleration vectors to use.
- *sunctx* – the *SUNContext* object (see §4.2)

Return value: A SUNNonlinSol object if the constructor exits successfully, otherwise it will be `NULL`.

Since the accelerated fixed point iteration (9.6) does not require the setup or solution of any linear systems, the SUNNonlinSol_FixedPoint module implements all of the functions defined in §9.1.1–§9.1.3 except for the *SUNNonlinSolSetup()*, *SUNNonlinSolSetLSetupFn()*, and *SUNNonlinSolSetLSolveFn()* functions, that are set to `NULL`. The SUNNonlinSol_FixedPoint functions have the same names as those defined by the generic SUNNonlinSol API

with `_FixedPoint` appended to the function name. Unless using the SUNNonlinSol_FixedPoint module as a standalone nonlinear solver the generic functions defined in §9.1.1–§9.1.3 should be called in favor of the SUNNonlinSol_FixedPoint-specific implementations.

The SUNNonlinSol_FixedPoint module also defines the following user-callable functions.

int **SUNNonlinSolGetSysFn_FixedPoint**(*SUNNonlinearSolver* NLS, *SUNNonlinSolSysFn* *SysFn)
This returns the fixed-point function that defines the nonlinear system.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *SysFn* – the function defining the nonlinear system.

Return value: The return value is zero for a successful call, and a negative value for a failure.

Notes: This function is intended for users that wish to evaluate the fixed-point function in a custom convergence test function for the SUNNonlinSol_FixedPoint module. We note that SUNNonlinSol_FixedPoint will not leverage the results from any user calls to *SysFn*.

int **SUNNonlinSolSetDamping_FixedPoint**(*SUNNonlinearSolver* NLS, *realtype* beta)
This sets the damping parameter β to use with Anderson acceleration. By default damping is disabled i.e., $\beta = 1.0$.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *beta* – the damping parameter $0 < \beta \leq 1$.

Return value:

- `SUN_NLS_SUCCESS` if successful.
- `SUN_NLS_MEM_NULL` if `NLS` was `NULL`.
- `SUN_NLS_ILL_INPUT` if `beta` was negative.

Notes: A `beta` value should satisfy $0 < \beta < 1$ if damping is to be used. A value of one or more will disable damping.

int **SUNNonlinSolSetInfoFile_FixedPoint**(*SUNNonlinearSolver* NLS, FILE *info_file)
Thissets the output file where all informative (non-error) messages should be directed.

Arguments:

- *NLS* – a SUNNonlinSol object.
- ***info_file* – pointer to output file (`stdout` by default);** a `NULL` input will disable output.

Return value:

- `SUN_NLS_SUCCESS` if successful.
- `SUN_NLS_MEM_NULL` if `NLS` was `NULL`.
- `SUN_NLS_ILL_INPUT` if SUNDIALS was not built with monitoring enabled.

Notes: This function is intended for users that wish to monitor the nonlinear solver progress. By default, the file pointer is set to `stdout`.

Warning: SUNDIALS must be built with the CMake option `SUNDIALS_BUILD_WITH_MONITORING` to utilize this function. See §11.1.2 for more information.

int **SUNNonlinSolSetPrintLevel_FixedPoint**(*SUNNonlinearSolver* NLS, int print_level)

This specifies the level of verbosity of the output.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *print_level* – flag indicating level of verbosity; must be one of:
  - 0, no information is printed (default).
  - 1, for each nonlinear iteration the residual norm is printed.

Return value:

- `SUN_NLS_SUCCESS` if successful.
- `SUN_NLS_MEM_NULL` if `NLS` was `NULL`.
- `SUN_NLS_ILL_INPUT` if SUNDIALS was not built with monitoring enabled, or the print level value was invalid.

Notes: This function is intended for users that wish to monitor the nonlinear solver progress. By default, the print level is 0.

> **Warning:** SUNDIALS must be built with the CMake option `SUNDIALS_BUILD_WITH_MONITORING` to utilize this function. See §11.1.2 for more information.

9.4.3 SUNNonlinSol_FixedPoint content

The *content* field of the SUNNonlinSol_FixedPoint module is the following structure.

```
struct _SUNNonlinearSolverContent_FixedPoint {

  SUNNonlinSolSysFn      Sys;
  SUNNonlinSolConvTestFn CTest;

  int          m;
  int         *imap;
  realtype    *R;
  booleantype  damping
  realtype     beta
  realtype    *gamma;
  realtype    *cvals;
  N_Vector    *df;
  N_Vector    *dg;
  N_Vector    *q;
  N_Vector    *Xvecs;
  N_Vector     yprev;
  N_Vector     gy;
  N_Vector     fold;
  N_Vector     gold;
  N_Vector     delta;
  int          curiter;
  int          maxiters;
  long int     niters;
```

(continues on next page)

(continued from previous page)

```
  long int     nconvfails;
  void        *ctest_data;
  int          print_level;
  FILE*        info_file;
};
```

The following entries of the *content* field are always allocated:

- `Sys` – function for evaluating the nonlinear system,
- `CTest` – function for checking convergence of the fixed point iteration,
- `yprev` – `N_Vector` used to store previous fixed-point iterate,
- `gy` – `N_Vector` used to store $G(y)$ in fixed-point algorithm,
- `delta` – `N_Vector` used to store difference between successive fixed-point iterates,
- `curiter` – the current number of iterations in the solve attempt,
- `maxiters` – the maximum number of fixed-point iterations allowed in a solve,
- `niters` – the total number of nonlinear iterations across all solves,
- `nconvfails` – the total number of nonlinear convergence failures across all solves,
- `ctest_data` – the data pointer passed to the convergence test function,
- `m` – number of acceleration vectors,
- `print_level` - controls the amount of information to be printed to the info file, and
- `info_file` - the file where all informative (non-error) messages will be directed.

If Anderson acceleration is requested (i.e., $m > 0$ in the call to *SUNNonlinSol_FixedPoint()*), then the following items are also allocated within the *content* field:

- `imap` – index array used in acceleration algorithm (length `m`),
- `damping` – a flag indicating if damping is enabled,
- `beta` – the damping parameter,
- `R` – small matrix used in acceleration algorithm (length `m*m`),
- `gamma` – small vector used in acceleration algorithm (length `m`),
- `cvals` – small vector used in acceleration algorithm (length `m+1`),
- `df` – array of `N_Vectors` used in acceleration algorithm (length `m`),
- `dg` – array of `N_Vectors` used in acceleration algorithm (length `m`),
- `q` – array of `N_Vectors` used in acceleration algorithm (length `m`),
- `Xvecs` – `N_Vector` pointer array used in acceleration algorithm (length `m+1`),
- `fold` – `N_Vector` used in acceleration algorithm, and
- `gold` – `N_Vector` used in acceleration algorithm.

9.5 The SUNNonlinSol_PetscSNES implementation

This section describes the SUNNonlinSol interface to the PETSc SNES nonlinear solver(s). To enable the SUNNonlinSol_PetscSNES module, SUNDIALS must be configured to use PETSc. Instructions on how to do this are given in §11.1.4.5. To access the SUNNonlinSol_PetscSNES module, include the header file `sunnonlinsol/sunnonlinsol_petscsnes.h`. The library to link to is `libsundials_sunnonlinsolpetsc.lib` where `.lib` is typically `.so` for shared libaries and `.a` for static libraries. Users of the SUNNonlinSol_PetscSNES module should also see §6.9 which discusses the NVECTOR interface to the PETSc `Vec` API.

9.5.1 SUNNonlinSol_PetscSNES description

The SUNNonlinSol_PetscSNES implementation allows users to utilize a PETSc SNES nonlinear solver to solve the nonlinear systems that arise in the SUNDIALS integrators. Since SNES uses the KSP linear solver interface underneath it, the SUNNonlinSol_PetscSNES implementation does not interface with SUNDIALS linear solvers. Instead, users should set nonlinear solver options, linear solver options, and preconditioner options through the PETSc SNES, KSP, and PC APIs.

Important usage notes for the SUNNonlinSol_PetscSNES implementation:

- The SUNNonlinSol_PetscSNES implementation handles calling `SNESSetFunction` at construction. The actual residual function $F(y)$ is set by the SUNDIALS integrator when the SUNNonlinSol_PetscSNES object is attached to it. Therefore, a user should not call `SNESSetFunction` on a `SNES` object that is being used with SUNNonlinSol_PetscSNES. For these reasons it is recommended, although not always necessary, that the user calls *SUNNonlinSol_PetscSNES()* with the new `SNES` object immediately after calling `SNESCreate`.
- The number of nonlinear iterations is tracked by SUNDIALS separately from the count kept by SNES. As such, the function *SUNNonlinSolGetNumIters()* reports the cumulative number of iterations across the lifetime of the *SUNNonlinearSolver* object.
- Some "converged" and "diverged" convergence reasons returned by SNES are treated as recoverable convergence failures by SUNDIALS. Therefore, the count of convergence failures returned by *SUNNonlinSolGetNumConvFails()* will reflect the number of recoverable convergence failures as determined by SUNDIALS, and may differ from the count returned by `SNESGetNonlinearStepFailures`.
- The SUNNonlinSol_PetscSNES module is not currently compatible with the CVODES or IDAS staggered or simultaneous sensitivity strategies.

9.5.2 SUNNonlinearSolver_PetscSNES functions

The SUNNonlinSol_PetscSNES module provides the following constructor for creating a *SUNNonlinearSolver* object.

SUNNonlinearSolver **SUNNonlinSol_PetscSNES**(*N_Vector* y, SNES snes, *SUNContext* sunctx)

This creates a SUNNonlinSol object that wraps a PETSc `SNES` object for use with SUNDIALS. This will call `SNESSetFunction` on the provided `SNES` object.

Arguments:

- *snes* – a PETSc `SNES` object.
- *y* – a `N_Vector` object of type NVECTOR_PETSC that is used as a template for the residual vector.
- *sunctx* – the *SUNContext* object (see §4.2)

Return value: A SUNNonlinSol object if the constructor exits successfully, otherwise it will be `NULL`.

> **Warning:** This function calls `SNESSetFunction` and will overwrite whatever function was previously set. Users should not call `SNESSetFunction` on the `SNES` object provided to the constructor.

The SUNNonlinSol_PetscSNES module implements all of the functions defined in §9.1.1–§9.1.3 except for *SUNNonlinSolSetup()*, *SUNNonlinSolSetLSetupFn()*, *SUNNonlinSolSetLSolveFn()*, *SUNNonlinSolSetConvTestFn()*, and *SUNNonlinSolSetMaxIters()*.

The SUNNonlinSol_PetscSNES functions have the same names as those defined by the generic SUNNonlinSol API with `_PetscSNES` appended to the function name. Unless using the SUNNonlinSol_PetscSNES module as a standalone nonlinear solver the generic functions defined in §9.1.1–§9.1.3 should be called in favor of the SUNNonlinSol_PetscSNES specific implementations.

The SUNNonlinSol_PetscSNES module also defines the following user-callable functions.

int **SUNNonlinSolGetSNES_PetscSNES**(*SUNNonlinearSolver* NLS, SNES *snes)
This gets the `SNES` object that was wrapped.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *snes* – a pointer to a PETSc `SNES` object that will be set upon return.

Return value: The return value (of type `int`) should be zero for a successful call, and a negative value for a failure.

int **SUNNonlinSolGetPetscError_PetscSNES**(*SUNNonlinearSolver* NLS, PestcErrorCode *error)
This gets the last error code returned by the last internal call to a PETSc API function.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *error* – a pointer to a PETSc error integer that will be set upon return.

Return value: The return value (of type `int`) should be zero for a successful call, and a negative value for a failure.

int **SUNNonlinSolGetSysFn_PetscSNES**(*SUNNonlinearSolver* NLS, *SUNNonlinSolSysFn* *SysFn)
This returns the residual function that defines the nonlinear system.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *SysFn* – the function defining the nonlinear system.

Return value: The return value (of type `int`) should be zero for a successful call, and a negative value for a failure.

9.5.3 SUNNonlinearSolver_PetscSNES content

The *content* field of the SUNNonlinSol_PetscSNES module is the following structure.

```
struct _SUNNonlinearSolverContent_PetscSNES {
  int sysfn_last_err;
  PetscErrorCode petsc_last_err;
  long int nconvfails;
  long int nni;
  void *imem;
```

(continues on next page)

(continued from previous page)

```
  SNES snes;
  Vec r;
  N_Vector y, f;
  SUNNonlinSolSysFn Sys;
};
```

These entries of the *content* field contain the following information:

- `sysfn_last_err` – last error returned by the system defining function,
- `petsc_last_err` – last error returned by PETSc,
- `nconvfails` – number of nonlinear converge failures (recoverable or not),
- `nni` – number of nonlinear iterations,
- `imem` – SUNDIALS integrator memory,
- `snes` – PETSc `SNES` object,
- `r` – the nonlinear residual,
- `y` – wrapper for PETSc vectors used in the system function,
- `f` – wrapper for PETSc vectors used in the system function,
- `Sys` – nonlinear system definining function.

Chapter 10

Tools for Memory Management

To support applications which leverage memory pools, or utilize a memory abstraction layer, SUNDIALS provides a set of utilities that we collectively refer to as the SUNMemoryHelper API. The goal of this API is to allow users to leverage operations defined by native SUNDIALS data structures while allowing the user to have finer-grained control of the memory management.

10.1 The SUNMemoryHelper API

This API consists of three new SUNDIALS types: *SUNMemoryType*, *SUNMemory*, and *SUNMemoryHelper*:

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);
};
```

10.1.1 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).

10.1.2 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).

10.1.3 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.

10.1.4 Implementing a custom SUNMemoryHelper

A particular implementation of the SUNMemoryHelper API must:

- Define and implement the required operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one SUNMemoryHelper module in the same code.
- Optionally, specify the *content* field of SUNMemoryHelper.
- Optionally, define and implement additional user-callable routines acting on the newly defined SUNMemoryHelper.

An example of a custom SUNMemoryHelper is given in examples/utilities/custom_memory_helper.h.

10.2 The SUNMemoryHelper_Cuda Implementation

The SUNMemoryHelper_Cuda module is an implementation of the SUNMemoryHelper API that interfaces to the NVIDIA [52] 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.

10.2.1 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).

10.3 The SUNMemoryHelper_Hip Implementation

The SUNMemoryHelper_Hip module is an implementation of the SUNMemoryHelper API that interfaces to the AMD ROCm HIP library [49]. 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.

10.3.1 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).

10.4 The SUNMemoryHelper_Sycl Implementation

The SUNMemoryHelper_Sycl module is an implementation of the `SUNMemoryHelper` API that interfaces to the SYCL abstraction layer. The implementation defines the constructor

SUNMemoryHelper **SUNMemoryHelper_Sycl**(*SUNContext* sunctx)

Allocates and returns a `SUNMemoryHelper` object for handling SYCL memory using the provided queue. Otherwise it returns `NULL`.

10.4.1 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 11

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 §11.1 and §11.2.

11.1 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.

11.1.1 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
```

11.1.1.1 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
```

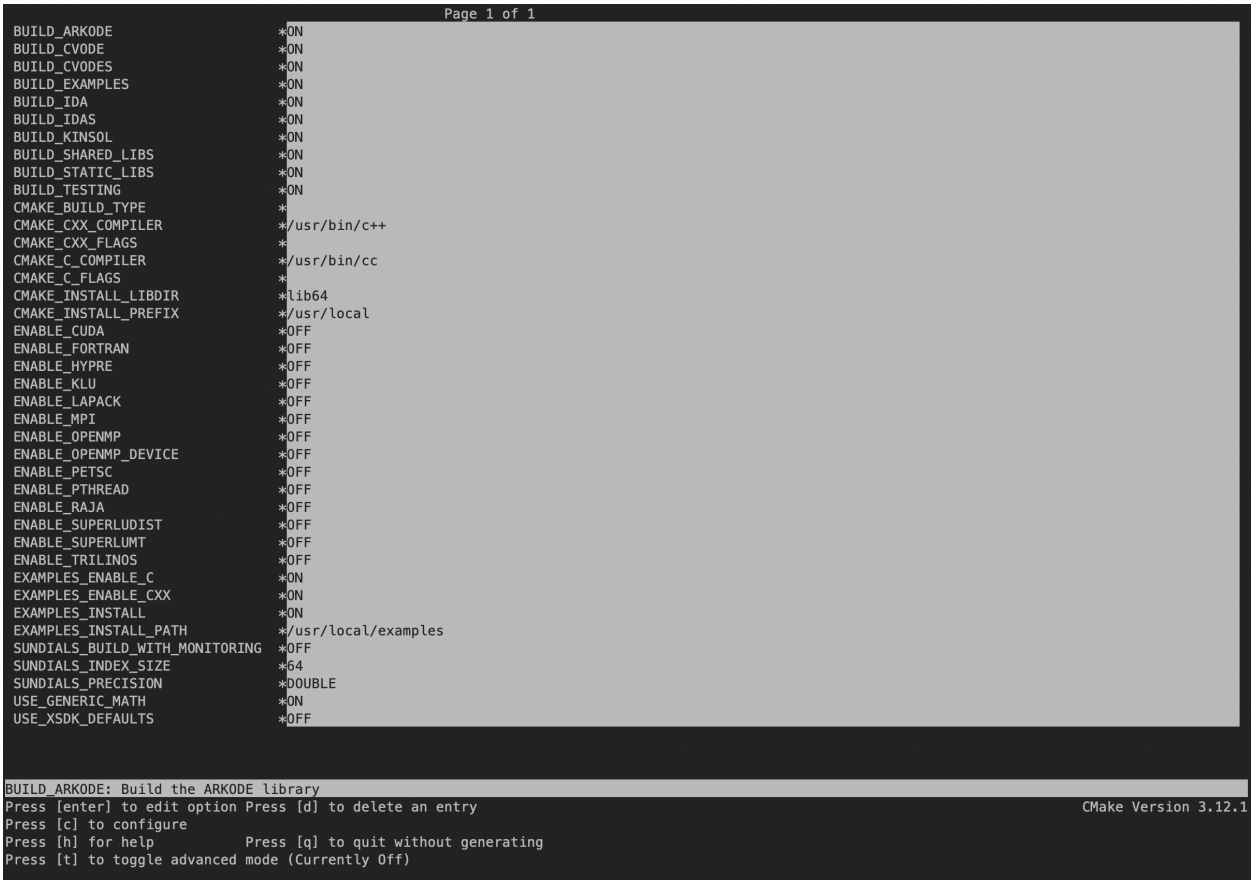

Fig. 11.1: Default configuration screen. Note: Initial screen is empty. To get this default configuration, press 'c' repeatedly (accepting default values denoted with asterisk) until the 'g' option is available.

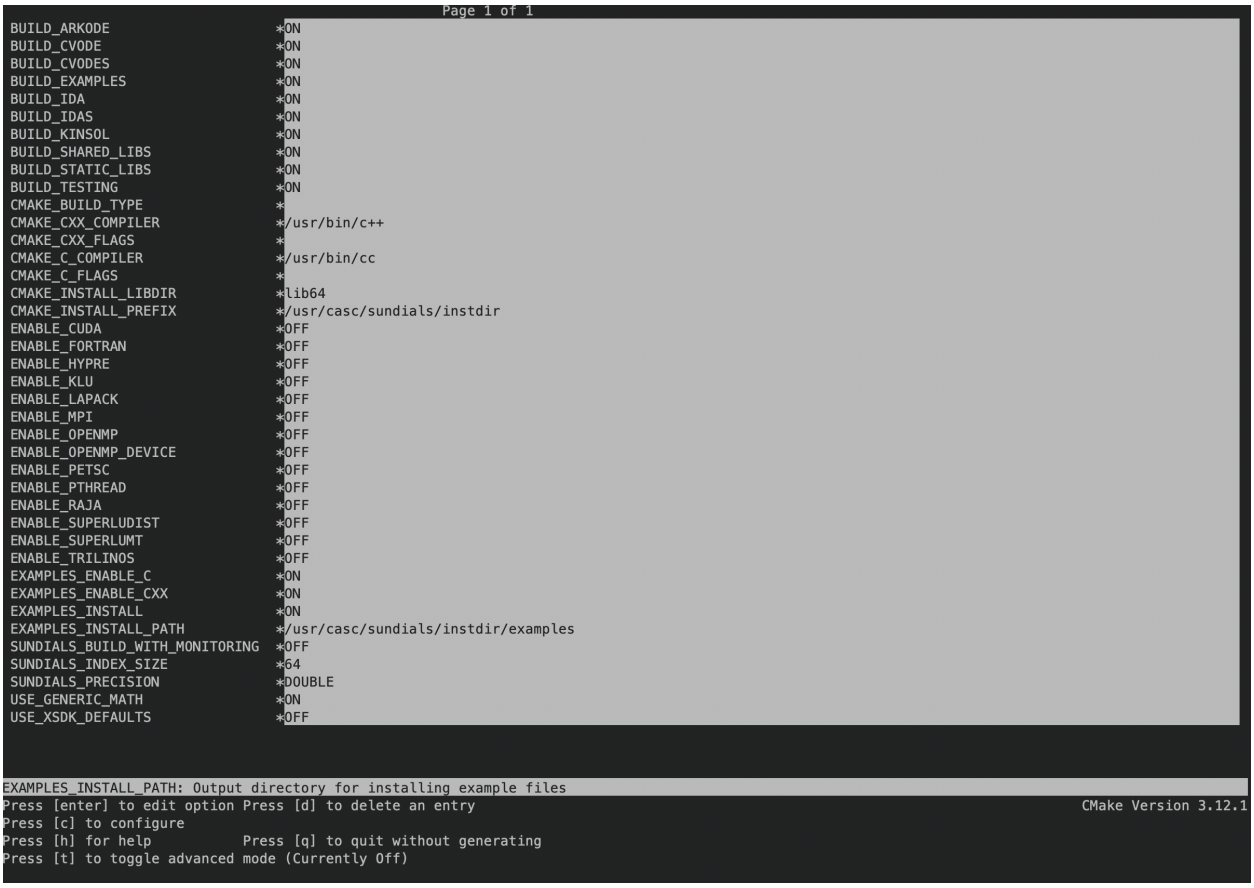

Fig. 11.2: Changing the INSTDIR for SUNDIALS and corresponding EXAMPLES.

11.1.1.2 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
```

11.1.2 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_CXX_COMPILER
C++ compiler

Default: `/usr/bin/c++`

Note: A C++ compiler (and all related options) are only are triggered if C++ examples are enabled (`EXAMPLES_ENABLE_CXX` is ON). All SUNDIALS solvers can be used from C++ applications by default without setting any additional configuration options.

CMAKE_CXX_FLAGS
Flags for C++ compiler

Default:

CMAKE_CXX_FLAGS_DEBUG
Flags used by the C++ compiler during debug builds

Default: `-g`

CMAKE_CXX_FLAGS_MINSIZEREL
Flags used by the C++ compiler during release minsize builds

Default: `-Os -DNDEBUG`

CMAKE_CXX_FLAGS_RELEASE
Flags used by the C++ compiler during release builds

Default: `-O3 -DNDEBUG`

CMAKE_CXX_STANDARD
The C++ standard to build C++ parts of SUNDIALS with.

Default: 11

Note: Options are 98, 11, 14, 17, 20. This option is only used when a C++ compiler is required.

CMAKE_Fortran_COMPILER
Fortran compiler

Default: /usr/bin/gfortran

Note: Fortran support (and all related options) are triggered only if either Fortran-C support (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 §11.1.4.

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 §11.1.4.

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 §11.1.4.

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 §11.1.4.

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

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 §11.1.4.

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 §11.1.4.

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 §11.1.4.

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

Note: 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

11.1.3 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
```

11.1.4 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.

11.1.4.1 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.

11.1.4.2 Building with KLU

The KLU libraries are part of SuiteSparse, a suite of sparse matrix software, available from the Texas A&M University website: http://faculty.cse.tamu.edu/davis/suitesparse.html .

SUNDIALS has been tested with SuiteSparse version 5.7.2. To enable KLU, set `ENABLE_KLU` to `ON`, set `KLU_INCLUDE_DIR` to the `include` path of the KLU installation and set `KLU_LIBRARY_DIR` to the `lib` path of the KLU installation. The CMake configure will result in populating the following variables: `AMD_LIBRARY`, `AMD_LIBRARY_DIR`, `BTF_LIBRARY`, `BTF_LIBRARY_DIR`, `COLAMD_LIBRARY`, `COLAMD_LIBRARY_DIR`, and `KLU_LIBRARY`.

11.1.4.3 Building with SuperLU_DIST

The SuperLU_DIST libraries are available for download from the Lawrence Berkeley National Laboratory website: https://portal.nersc.gov/project/sparse/superlu/#superlu_dist.

SUNDIALS has been tested with SuperLU_DIST 6.1.1. To enable SuperLU_DIST, set `ENABLE_SUPERLUDIST` to `ON`, set `SUPERLUDIST_INCLUDE_DIR` to the `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.

11.1.4.4 Building with SuperLU_MT

The SuperLU_MT libraries are available for download from the Lawrence Berkeley National Laboratory website: https://portal.nersc.gov/project/sparse/superlu/#superlu_mt.

SUNDIALS has been tested with SuperLU_MT version 3.1. To enable SuperLU_MT, set `ENABLE_SUPERLUMT` to `ON`, set `SUPERLUMT_INCLUDE_DIR` to the `SRC` path of the SuperLU_MT installation, and set the variable `SUPERLUMT_LIBRARY_DIR` to the `lib` path of the SuperLU_MT installation. At the same time, the variable `SUPERLUMT_LIBRARIES` must be set to a semi-colon separated list of other libraries SuperLU_MT depends on. For example, if SuperLU_MT was build with an external blas library, then include the full path to the blas library in this list. Additionally, the variable `SUPERLUMT_THREAD_TYPE` must be set to either `Pthread` or `OpenMP`.

Do not mix thread types when building SUNDIALS solvers. If threading is enabled for SUNDIALS by having either `ENABLE_OPENMP` or `ENABLE_PTHREAD` set to `ON` then SuperLU_MT should be set to use the same threading type.

11.1.4.5 Building with PETSc

The PETSc libraries are available for download from the Argonne National Laboratory website: http://www.mcs.anl.gov/petsc .

SUNDIALS has been tested with PETSc versions 3.10.0 – 3.14.0. 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`.

11.1.4.6 Building with *hypre*

The *hypre* libraries are available for download from the Lawrence Livermore National Laboratory website: http://computing.llnl.gov/projects/hypre. SUNDIALS has been tested with *hypre* version 2.19.0. To enable *hypre*, set `ENABLE_HYPRE` to `ON`, set `HYPRE_INCLUDE_DIR` to the `include` path of the *hypre* installation, and set the variable `HYPRE_LIBRARY_DIR` to the `lib` path of the *hypre* installation.

Note: SUNDIALS must be configured so that `SUNDIALS_INDEX_SIZE` (or equivalently, `XSDK_INDEX_SIZE`) equals the precision of `HYPRE_BigInt` in the corresponding *hypre* installation.

11.1.4.7 Building with Magma

The Magma libraries are ....

SUNDIALS has been tested with Magma version ...

11.1.4.8 Building with OneMKL

The OneMKL libraries are ....

SUNDIALS has been tested with OneMKL version ...

11.1.4.9 Building with CUDA

SUNDIALS CUDA modules and examples have been tested with version 10 and 11 of the CUDA toolkit. To build them, you need to install the Toolkit and compatible NVIDIA drivers. Both are available for download from the NVIDIA website: https://developer.nvidia.com/cuda-downloads. To enable CUDA, set `ENABLE_CUDA` to `ON`. If CUDA is installed in a nonstandard location, you may be prompted to set the variable `CUDA_TOOLKIT_ROOT_DIR` with your CUDA Toolkit installation path. To enable CUDA examples, set `EXAMPLES_ENABLE_CUDA` to `ON`.

11.1.4.10 Building with RAJA

RAJA is a performance portability layer developed by Lawrence Livermore National Laboratory and can be obtained from https://github.com/LLNL/RAJA. SUNDIALS RAJA modules and examples have been tested with RAJA version 0.14.0. Building SUNDIALS RAJA modules requires a CUDA, HIP, or SYCL enabled RAJA installation. To enable RAJA, set `ENABLE_RAJA` to `ON`, set `SUNDIALS_RAJA_BACKENDS` to the desired backend (`CUDA`, `HIP`, or `SYCL`), and set `ENABLE_CUDA`, `ENABLE_HIP`, or `ENABLE_SYCL` to `ON` depending on the selected backend. If RAJA is installed in a nonstandard location you will be prompted to set the variable `RAJA_DIR` with the path to the RAJA CMake configuration file. To enable building the RAJA examples set `EXAMPLES_ENABLE_CXX` to `ON`.

11.1.4.11 Building with XBraid

The XBraid library is available for download from the XBraid GitHub: https://github.com/XBraid/xbraid. SUNDIALS has been tested with XBraid version 3.0.0. To enable XBraid, 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`.

11.1.5 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
```

11.1.6 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.

11.1.7 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.

11.2 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.

11.2.1 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 11.1: 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
		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

continues on next page

Table 11.1 – continued from previous page

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
	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

continues on next page

Table 11.1 – continued from previous page

	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
	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

continues on next page

Table 11.1 – continued from previous page

		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
		kinsol/kinsol_impl.h
		kinsol/kinsol_ls.h
		kinsol/kinsol_spils.h

Chapter 12

IDAS 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.

12.1 IDAS input constants

Table 12.1: IDAS Input Constants

IDAS main solver module		
IDA_NORMAL	1	Solver returns at specified output time.
IDA_ONE_STEP	2	Solver returns after each successful step.
IDA_SIMULTANEOUS	1	Simultaneous corrector forward sensitivity method.
IDA_STAGGERED	2	Staggered corrector forward sensitivity method.
IDA_CENTERED	1	Central difference quotient approximation (2^{nd} order) of the sensitivity RHS.
IDA_FORWARD	2	Forward difference quotient approximation (1^{st} order) of the sensitivity RHS.
IDA_YA_YDP_INIT	1	Compute y_a and $\dot{y}_d$, given y_d.
IDA_Y_INIT	2	Compute y, given $\dot{y}$.
IDAS adjoint solver module		
IDA_HERMITE	1	Use Hermite interpolation.
IDA_POLYNOMIAL	2	Use variable-degree polynomial interpolation.
Iterative linear solver module		
SUN_PREC_NONE	0	No preconditioning
SUN_PREC_LEFT	1	Preconditioning on the left.
SUN_MODIFIED_GS	1	Use modified Gram-Schmidt procedure.
SUN_CLASSICAL_GS	2	Use classical Gram-Schmidt procedure.

12.2 IDAS output constants

Table 12.2: IDAS Output Constants

IDAS main solver module		
IDA_SUCCESS	0	Successful function return.
IDA_TSTOP_RETURN	1	IDASolve succeeded by reaching the specified stopping point.
IDA_ROOT_RETURN	2	IDASolve succeeded and found one or more roots.
IDA_WARNING	99	IDASolve succeeded but an unusual situation occurred.
IDA_TOO_MUCH_WORK	-1	The solver took mxstep internal steps but could not reach tout.
IDA_TOO_MUCH_ACC	-2	The solver could not satisfy the accuracy demanded by the user for some internal step.
IDA_ERR_FAIL	-3	Error test failures occurred too many times during one internal time step or minimum step size was reached.
IDA_CONV_FAIL	-4	Convergence test failures occurred too many times during one internal time step or minimum step size was reached.
IDA_LINIT_FAIL	-5	The linear solver's initialization function failed.
IDA_LSETUP_FAIL	-6	The linear solver's setup function failed in an unrecoverable manner.
IDA_LSOLVE_FAIL	-7	The linear solver's solve function failed in an unrecoverable manner.
IDA_RES_FAIL	-8	The user-provided residual function failed in an unrecoverable manner.
IDA_REP_RES_FAIL	-9	The user-provided residual function repeatedly returned a recoverable error flag, but the solver was unable to recover.
IDA_RTFUNC_FAIL	-10	The rootfinding function failed in an unrecoverable manner.
IDA_CONSTR_FAIL	-11	The inequality constraints were violated and the solver was unable to recover.
IDA_FIRST_RES_FAIL	-12	The user-provided residual function failed recoverably on the first call.
IDA_LINESEARCH_FAIL	-13	The line search failed.
IDA_NO_RECOVERY	-14	The residual function, linear solver setup function, or linear solver solve function had a recoverable failure, but IDACalcIC could not recover.
IDA_NLS_INIT_FAIL	-15	The nonlinear solver's init routine failed.
IDA_NLS_SETUP_FAIL	-16	The nonlinear solver's setup routine failed.
IDA_MEM_NULL	-20	The ida_mem argument was NULL.
IDA_MEM_FAIL	-21	A memory allocation failed.
IDA_ILL_INPUT	-22	One of the function inputs is illegal.
IDA_NO_MALLOC	-23	The IDAS memory was not allocated by a call to IDAInit.
IDA_BAD_EWT	-24	Zero value of some error weight component.
IDA_BAD_K	-25	The k-th derivative is not available.
IDA_BAD_T	-26	The time t is outside the last step taken.
IDA_BAD_DKY	-27	The vector argument where derivative should be stored is NULL.
IDA_NO_QUAD	-30	Quadratures were not initialized.
IDA_QRHS_FAIL	-31	The user-provided right-hand side function for quadratures failed in an unrecoverable manner.
IDA_FIRST_QRHS_ERR	-32	The user-provided right-hand side function for quadratures failed in an unrecoverable manner on the first call.
IDA_REP_QRHS_ERR	-33	The user-provided right-hand side repeatedly returned a recoverable error flag, but the solver was unable to recover.
IDA_NO_SENS	-40	Sensitivities were not initialized.

continues on next page

Table 12.2 – continued from previous page

IDAS main solver module		
IDA_SRES_FAIL	-41	The user-provided sensitivity residual function failed in an unrecoverable manner.
IDA_REP_SRES_ERR	-42	The user-provided sensitivity residual function repeatedly returned a recoverable error flag, but the solver was unable to recover.
IDA_BAD_IS	-43	The sensitivity identifier is not valid.
IDA_NO_QUADSENS	-50	Sensitivity-dependent quadratures were not initialized.
IDA_QSRHS_FAIL	-51	The user-provided sensitivity-dependent quadrature right-hand side function failed in an unrecoverable manner.
IDA_FIRST_QSRHS_ERR	-52	The user-provided sensitivity-dependent quadrature right-hand side function failed in an unrecoverable manner on the first call.
IDA_REP_QSRHS_ERR	-53	The user-provided sensitivity-dependent quadrature right-hand side repeatedly returned a recoverable error flag, but the solver was unable to recover.
IDAS adjoint solver module		
IDA_NO_ADJ	-101	The combined forward-backward problem has not been initialized.
IDA_NO_FWD	-102	IDASolveF has not been previously called.
IDA_NO_BCK	-103	No backward problem was specified.
IDA_BAD_TB0	-104	The desired output for backward problem is outside the interval over which the forward problem was solved.
IDA_REIFWD_FAIL	-105	No checkpoint is available for this hot start.
IDA_FWD_FAIL	-106	IDASolveB failed because IDASolve was unable to store data between two consecutive checkpoints.
IDA_GETY_BADT	-107	Wrong time in interpolation function.
IDALS linear solver interface		
IDALS_SUCCESS	0	Successful function return.
IDALS_MEM_NULL	-1	The ida_mem argument was NULL.
IDALS_LMEM_NULL	-2	The IDALS linear solver has not been initialized.
IDALS_ILL_INPUT	-3	The IDALS solver is not compatible with the current N_Vector module, or an input value was illegal.
IDALS_MEM_FAIL	-4	A memory allocation request failed.
IDALS_PMEM_NULL	-5	The preconditioner module has not been initialized.
IDALS_JACFUNC_UNRECVR	-6	The Jacobian function failed in an unrecoverable manner.
IDALS_JACFUNC_RECVR	-7	The Jacobian function had a recoverable error.
IDALS_SUNMAT_FAIL	-8	An error occurred with the current SUNMatrix module.
IDALS_SUNLS_FAIL	-9	An error occurred with the current SUNLinearSolver module.
IDALS_NO_ADJ	-101	The combined forward-backward problem has not been initialized.
IDALS_LMEMB_NULL	-102	The linear solver was not initialized for the backward phase.

Chapter 13

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 13.1 – 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] K. E. Brenan, S. L. Campbell, and L. R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. SIAM, Philadelphia, Pa, 1996.

[5] P. N. Brown. A local convergence theory for combined inexact-Newton/finite difference projection methods. *SIAM J. Numer. Anal.*, 24(2):407–434, 1987.

[6] P. N. Brown, G. D. Byrne, and A. C. Hindmarsh. VODE, a Variable-Coefficient ODE Solver. *SIAM J. Sci. Stat. Comput.*, 10:1038–1051, 1989.

[7] P. N. Brown and A. C. Hindmarsh. Reduced Storage Matrix Methods in Stiff ODE Systems. *J. Appl. Math. & Comp.*, 31:49–91, 1989.

[8] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold. Using Krylov Methods in the Solution of Large-Scale Differential-Algebraic Systems. *SIAM J. Sci. Comput.*, 15:1467–1488, 1994.

[9] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold. Consistent Initial Condition Calculation for Differential-Algebraic Systems. *SIAM J. Sci. Comput.*, 19:1495–1512, 1998.

[10] P. N. Brown and Y. Saad. Hybrid Krylov Methods for Nonlinear Systems of Equations. *SIAM J. Sci. Stat. Comput.*, 11:450–481, 1990.

[11] 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.

[12] G. D. Byrne and A. C. Hindmarsh. User Documentation for PVODE, An ODE Solver for Parallel Computers. Technical Report UCRL-ID-130884, LLNL, May 1998.

[13] G. D. Byrne and A. C. Hindmarsh. PVODE, An ODE Solver for Parallel Computers. *Intl. J. High Perf. Comput. Apps.*, 13(4):254–365, 1999.

[14] Y. Cao, S. Li, L. R. Petzold, and R. Serban. Adjoint Sensitivity Analysis for Differential-Algebraic Equations: The Adjoint DAE System and its Numerical Solution. *SIAM J. Sci. Comput.*, 24(3):1076–1089, 2003.

[15] M. Caracotsios and W. E. Stewart. Sensitivity Analysis of Initial Value Problems with Mixed ODEs and Algebraic Equations. *Computers and Chemical Engineering*, 9:359–365, 1985.

[16] S. D. Cohen and A. C. Hindmarsh. \mbox CVODE, a Stiff/Nonstiff ODE Solver in C. *Computers in Physics*, 10(2):138–143, 1996.

[17] A. M. Collier, A. C. Hindmarsh, R. Serban, and C.S. Woodward. User Documentation for KINSOL v6.0.0. Technical Report UCRL-SM-208116, LLNL, 2021.

[18] T. A. Davis and P. N. Ekanathan. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Softw.*, 2010.

[19] R. S. Dembo, S. C. Eisenstat, and T. Steihaug. Inexact Newton Methods. *SIAM J. Numer. Anal.*, 19:400–408, 1982.

[20] 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.

[21] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM, Philadelphia, 1996.

[22] 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.

[23] H. Fang and Y. Saad. Two classes of secant methods for nonlinear acceleration. *Numer. Linear Algebra Appl.*, 16:197–221, 2009.

[24] W. F. Feehery, J. E. Tolsma, and P. I. Barton. Efficient Sensitivity Analysis of Large-Scale Differential-Algebraic Systems. *Applied Numer. Math.*, 25(1):41–54, 1997.

[25] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14:470–482, 1993.

[26] 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.

[27] 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.

[28] K. L. Hiebert and L. F. Shampine. Implicitly Defined Output Points for Solutions of ODEs. Technical Report SAND80-0180, Sandia National Laboratories, February 1980.

[29] A. C. Hindmarsh. The PVODE and IDA Algorithms. Technical Report UCRL-ID-141558, LLNL, December 2000.

[30] 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.

[31] A. C. Hindmarsh and R. Serban. User Documentation for CVODE v6.0.0. Technical Report UCRL-SM-208108, LLNL, 2021.

[32] A. C. Hindmarsh, R. Serban, and A. Collier. Example Programs for IDA v6.0.0. Technical Report UCRL-SM-208113, LLNL, 2021.

[33] A. C. Hindmarsh, R. Serban, and D. R. Reynolds. Example Programs for CVODE v6.0.0. Technical Report, LLNL, 2021. UCRL-SM-208110.

[34] A. C. Hindmarsh and A. G. Taylor. PVODE and KINSOL: Parallel Software for Differential and Nonlinear Systems. Technical Report UCRL-ID-129739, LLNL, February 1998.

[35] 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.

[36] C. T. Kelley. *Iterative Methods for Solving Linear and Nonlinear Equations*. SIAM, Philadelphia, 1995.

[37] S. Li, L. R. Petzold, and W. Zhu. Sensitivity Analysis of Differential-Algebraic Equations: A Comparison of Methods on a Special Problem. *Applied Num. Math.*, 32:161–174, 2000.

[38] X. S. Li. An overview of SuperLU: algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.

[39] 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.

[40] 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.

[41] 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.

[42] T. Maly and L. R. Petzold. Numerical Methods and Software for Sensitivity Analysis of Differential-Algebraic Systems. *Applied Numerical Mathematics*, 20:57–79, 1997.

[43] D.B. Ozyurt and P.I. Barton. Cheap second order directional derivatives of stiff ODE embedded functionals. *SIAM J. of Sci. Comp.*, 26(5):1725–1743, 2005.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] H. F. Walker and P. Ni. Anderson acceleration for fixed-point iterations. *SIAM Jour. Num. Anal.*, 49(4):1715–1735, 2011.

[49] N.a. AMD ROCm Documentation. https://rocmdocs.amd.com/en/latest/index.html.

[50] N.a. Intel oneAPI Programming Guide. https://software.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html.

[51] N.a. KLU Sparse Matrix Factorization Library. http://faculty.cse.tamu.edu/davis/suitesparse.html.

[52] N.a. NVIDIA CUDA Programming Guide. https://docs.nvidia.com/cuda/index.html.

[53] N.a. NVIDIA cuSOLVER Programming Guide. https://docs.nvidia.com/cuda/cusolver/index.html.

[54] N.a. NVIDIA cuSPARSE Programming Guide. https://docs.nvidia.com/cuda/cusparse/index.html.

[55] N.a. SuperLU_DIST Parallel Sparse Matrix Factorization Library. http://crd-legacy.lbl.gov/\protect\unhbox\voidb@x\protect\penalty\@M\xiaoye/SuperLU/.

[56] N.a. SuperLU_MT Threaded Sparse Matrix Factorization Library. http://crd-legacy.lbl.gov/\protect\unhbox\voidb@x\protect\penalty\@M\xiaoye/SuperLU/.

Index

K

L

M

N

P

R

S

U

V

X