# User Documentation for IDA v5.7.0 (SUNDIALS v5.7.0)

Alan C. Hindmarsh<sup>1</sup>, Radu Serban<sup>1</sup>, Cody J. Balos<sup>1</sup>, David J. Gardner<sup>1</sup>, Daniel R. Reynolds<sup>2</sup>, and Carol S. Woodward<sup>1</sup>

<sup>1</sup>Center for Applied Scientific Computing, Lawrence Livermore National Laboratory <sup>2</sup>Department of Mathematics, Southern Methodist University

February 1, 2021



#### DISCLAIMER

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

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

#### CONTRIBUTORS

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

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

## Contents

$\mathbf{Li}$	st of Tables	ix
Li	st of Figures	xi
1	Introduction1.1 Changes from previous versions1.2 Reading this User Guide1.3 SUNDIALS Release License	1 2 15 15
2	Mathematical Considerations2.1 IVP solution2.2 Preconditioning2.3 Rootfinding	19 19 23 23
3	Code Organization3.1 SUNDIALS organization3.2 IDA organization	25 25 26
4	Using IDA for C Applications 4.1 Access to library and header files 4.2 Data types 4.3 Header files 4.4 A skeleton of the user's main program 4.5 User-callable functions 4.6 User-supplied functions 4.7 A parallel band-block-diagonal preconditioner module	29 30 31 32 35 73 81
5	Using IDA for Fortran Applications 5.1 IDA Fortran 2003 Interface Module	87 87 93 93 93 103
6	IDA Features for GPU Accelerated Computing 6.1 SUNDIALS GPU Programming Model	
7	Description of the NVECTOR module 7.1 The NVECTOR API	134 136

	7.5	The NVECTOR_OPENMP implementation	146
	7.6	The NVECTOR_PTHREADS implementation	152
	7.7	The NVECTOR_PARHYP implementation	157
	7.8	The NVECTOR_PETSC implementation	161
	7.9	The NVECTOR_CUDA implementation	
	7.10	The NVECTOR_HIP implementation	
		The NVECTOR_RAJA implementation	
		The NVECTOR_SYCL implementation	
		The NVECTOR_OPENMPDEV implementation	
		The NVECTOR_TRILINOS implementation	
		The NVECTOR_MANYVECTOR implementation	
		The NVECTOR_MPIMANYVECTOR implementation	
		The NVECTOR_MPIPLUSX implementation	
		•	
	1.18	NVECTOR Examples	204
8		cription of the SUNMatrix module	209
	8.1	The SUNMatrix API	209
	8.2	SUNMatrix functions used by IDA	
	8.3	The SUNMatrix_Dense implementation	216
	8.4	The SUNMatrix_Band implementation	219
	8.5	The SUNMatrix_Sparse implementation	225
	8.6	The SUNMatrix_SLUNRloc implementation	232
	8.7	The SUNMatrix_cuSparse implementation	235
	8.8	The SUNMATRIX_MAGMADENSE implementation	240
9	Des	cription of the SUNLinearSolver module	245
•	9.1	The SUNLinearSolver API	
	9.2	Compatibility of SUNLinearSolver modules	
	9.3	Implementing a custom SUNLinearSolver module	
	9.4	IDA SUNLinearSolver interface	
	9.4 $9.5$	The SUNLinearSolver_Dense implementation	
	9.6	The SUNLinearSolver_Band implementation	
	9.7	The SUNLinearSolver_LapackDense implementation	
	9.8	The SUNLinearSolver_LapackBand implementation	
	9.9	The SUNLinearSolver_KLU implementation	
		The SUNLinearSolver_SuperLUDIST implementation	
		The SUNLinearSolver_SuperLUMT implementation	279
		The SUNLinearSolver_cuSolverSp_batchQR implementation	283
		The SUNLinearSolver_MagmaDense implementation	286
		The SUNLinearSolver_SPGMR implementation	287
		The SUNLinearSolver_SPFGMR implementation	295
	9.16	The SUNLinearSolver_SPBCGS implementation	302
		The SUNLinearSolver_SPTFQMR implementation	309
		The SUNLinear Solver_PCG implementation	315
	9.19	SUNLinearSolver Examples	321
10	Des	cription of the SUNNonlinearSolver module	323
_		The SUNNonlinearSolver API	323
		IDA SUNNonlinearSolver interface	334
		The SUNNonlinear Solver Newton implementation	337
		The SUNNonlinear Solver Petsc SNES implementation	
	10.1	The politional control is concerned implementation in the control of the control	0.11

11	Description of the SUNMemory module	345
	11.1 The SUNMemoryHelper API	345
	11.2 The SUNMemoryHelper_Cuda implementation	
	11.3 The SUNMemoryHelper_Hip implementation	
	11.4 The SUNMemoryHelper_Sycl implementation	
$\mathbf{A}$	SUNDIALS Package Installation Procedure	355
	A.1 CMake-based installation	356
	A.2 Building and Running Examples	
	A.3 Configuring, building, and installing on Windows	368
	A.4 Installed libraries and exported header files	
В	IDA Constants	377
	B.1 IDA input constants	377
	B.2 IDA output constants	
$\mathbf{C}$	SUNDIALS Release History	<b>37</b> 9
Bi	ibliography	381
In	$\operatorname{dex}$	385

## List of Tables

4.1	SUNDIALS linear solver interfaces and vector implementations that can be used for each.	35
$4.2 \\ 4.3$	Optional inputs for IDA and IDALS	45 60
5.1	Summary of Fortran 2003 interfaces for shared SUNDIALS modules	88
5.2	C/Fortran 2003 Equivalent Types	89
5.3	Keys for setting FIDA optional inputs	104
5.4	Description of the FIDA optional output arrays IOUT and ROUT	105
6.1	List of SUNDIALS GPU Enabled Modules	112
7.1	Vector Identifications associated with vector kernels supplied with SUNDIALS	130
7.2	List of vector functions usage by IDA code modules	135
8.1	Description of the SUNMatrix return codes	
8.2	Identifiers associated with matrix kernels supplied with SUNDIALS	213
8.3	SUNDIALS matrix interfaces and vector implementations that can be used for each	213
8.4	List of matrix functions usage by IDA code modules	215
9.1 9.2	Description of the SUNLinearSolver error codes	252
	each	255
9.3	List of linear solver function usage in the IDALS interface	258
10.1	Description of the SUNNonlinearSolver return codes	330
A.1	SUNDIALS libraries and header files	370
C.1	Release History	379

## List of Figures

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

## Chapter 1

### Introduction

IDA is part of a software family called SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic equation Solvers [32]. This suite consists of CVODE, ARKODE, KINSOL, and IDA, and variants of these with sensitivity analysis capabilities, CVODES and IDAS.

IDA is a general purpose solver for the initial value problem (IVP) for systems of differential-algebraic equations (DAEs). The name IDA stands for Implicit Differential-Algebraic solver. IDA is based on DASPK [14, 15], 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 IDA shares significant modules previously written within CASC at LLNL to support the ordinary differential equation (ODE) solvers CVODE [33, 20] and PVODE [18, 19], and also the nonlinear system solver KINSOL [21].

At present, IDA 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) [44], FGMRES (Flexible Generalized Minimum RESidual) [43], Bi-CGStab (Bi-Conjugate Gradient Stabilized) [46], TFQMR (Transpose-Free Quasi-Minimal Residual) [27], and PCG (Preconditioned Conjugate Gradient) [29] linear iterative methods. As Krylov methods, these require little matrix storage for solving the Newton equations as compared to direct methods. However, the algorithms allow for a user-supplied preconditioner matrix, and, for most problems, preconditioning is essential for an efficient solution.

For very large 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.

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

#### 1.1 Changes from previous versions

#### Changes in v5.7.0

A new NVECTOR implementation based on the SYCL abstraction layer has been added targeting Intel GPUs. At present the only SYCL compiler supported is the DPC++ (Intel oneAPI) compiler. See Section 7.12 for more details. This module is considered experimental and is subject to major changes even in minor releases.

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

#### Changes in v5.6.1

Fixed a bug in the SUNDIALS CMake which caused an error if the CMAKE\_CXX\_STANDARD and SUNDIALS\_RAJA\_BACKENDS options were not provided.

Fixed some compiler warnings when using the IBM XL compilers.

#### Changes in v5.6.0

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

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

A new optional operation, N\_VGetDeviceArrayPointer, was added to the N\_Vector API. This operation is useful for N\_Vectors that utilize dual memory spaces, e.g. the native SUNDIALS CUDA N\_Vector

The SUNMATRIX\_CUSPARSE and SUNLINEARSOLVER\_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.

#### Changes in v5.5.0

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

Added support for SuperLU DIST 6.3.0 or newer.

#### Changes in v5.4.0

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.

The expected behavior of SUNNonlinSolGetNumIters and SUNNonlinSolGetNumConvFails in the SUNNONLINSOL 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 SUNNONLINSOL 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 sections 6.1,11.1, 7.9 and 7.11 for more information.

The NVECTOR\_RAJA module has been updated to mirror the NVECTOR\_CUDA module. Notably, the update adds managed memory support to the NVECTOR\_RAJA module. Users of the module will need to update any calls to the N\_VMake\_Raja function because that signature was changed. This module remains experimental and is subject to change from version to version.

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

Added support for CUDA v11.

#### Changes in v5.3.0

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

Added 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 perfomance changes that they notice.

Added new capabilities for monitoring the solve phase in the SUNNONLINSOL\_NEWTON and SUNNONLINSOL\_FIXEDPOINT modules, and the SUNDIALS iterative linear solver modules. SUNDIALS must be built with the CMake option SUNDIALS\_BUILD\_WITH\_MONITORING to use these capabilities.

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

#### Changes in v5.2.0

Fixed a build system bug related to the Fortran 2003 interfaces when using the IBM XL compiler. When building the Fortran 2003 interfaces with an XL compiler it is recommended to set CMAKE\_Fortran\_COMPILER to f2003, xlf2003, or xlf2003\_r.

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

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

The function IDASetLinearSolutionScaling 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  $\alpha$  in the linear system matrix  $\frac{\partial F}{\partial y} + \alpha \frac{\partial F}{\partial \dot{y}}$ . Scaling is enabled by default when using a matrix-based linear solver.

#### Changes in v5.1.0

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

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

Fixed a build system bug related to finding PETSc when using the CMake variables PETSC\_INCLUDES and PETSC\_LIBRARIES instead of PETSC\_DIR.

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

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

#### Changes in v5.0.0

#### Build system changes

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

#### **NVECTOR** module changes

- Two new functions were added to aid in creating custom NVECTOR objects. The constructor N\_VNewEmpty allocates an "empty" generic NVECTOR with the object's content pointer and the function pointers in the operations structure initialized to NULL. When used in the constructor for custom objects this function will ease the introduction of any new optional operations to the NVECTOR API by ensuring only required operations need to be set. Additionally, the function N\_VCopyOps(w, v) has been added to copy the operation function pointers between vector objects. When used in clone routines for custom vector objects these functions also will ease the introduction of any new optional operations to the NVECTOR API by ensuring all operations are copied when cloning objects. See §7.1.6 for more details.
- Two new NVECTOR implementations, NVECTOR\_MANYVECTOR and NVECTOR\_MPIMANYVECTOR, have been created to support flexible partitioning of solution data among different processing elements (e.g., CPU + GPU) or for multi-physics problems that couple distinct MPI-based simulations together. This implementation is accompanied by additions to user documentation and SUNDIALS examples. See §7.15 and §7.16 for more details.
- One new required vector operation and ten new optional vector operations have been added to the NVECTOR API. The new required operation, N\_VGetLength, returns the global length of an N\_Vector. The optional operations have been added to support the new NVECTOR\_MPIMANYVECTOR implementation. The operation N\_VGetCommunicator must be implemented by subvectors that are combined to create an NVECTOR\_MPIMANYVECTOR, but is not used outside of this context. The remaining nine operations are optional local reduction operations intended to eliminate unnecessary latency when performing vector reduction operations (norms, etc.) on distributed memory systems. The optional local reduction vector operations are N\_VDotProdLocal, N\_VMaxNormLocal, N\_VMinLocal, N\_VL1NormLocal, N\_VWSqrSumLocal, N\_VWSqrSumLocal, N\_VWSqrSumMaskLocal, N\_VInvTestLocal, N\_VConstrMaskLocal, and N\_VMinQuotientLocal. If an NVECTOR implementation defines any of the local operations as NULL, then the NVECTOR\_MPIMANYVECTOR will call standard NVECTOR operations to complete the computation. See §7.1.4 for more details.
- An additional NVECTOR implementation, NVECTOR\_MPIPLUSX, has been created to support the MPI+X paradigm where X is a type of on-node parallelism (e.g., OpenMP, CUDA). The implementation is accompanied by additions to user documentation and SUNDIALS examples. See §7.17 for more details.
- The \*\_MPICuda and \*\_MPIRaja functions have been removed from the NVECTOR\_CUDA and NVECTOR\_RAJA implementations respectively. Accordingly, the nvector\_mpicuda.h,

nvector\_mpiraja.h, libsundials\_nvecmpicuda.lib, and libsundials\_nvecmpicudaraja.lib files have been removed. Users should use the NVECTOR\_MPIPLUSX module coupled in conjunction with the NVECTOR\_CUDA or NVECTOR\_RAJA modules to replace the functionality. The necessary changes are minimal and should require few code modifications. See the programs in examples/ida/mpicuda and examples/ida/mpiraja for examples of how to use the NVECTOR\_MPIPLUSX module with the NVECTOR\_CUDA and NVECTOR\_RAJA modules respectively.

- Fixed a memory leak in the NVECTOR\_PETSC module clone function.
- Made performance improvements to the NVECTOR\_CUDA module. Users who utilize a non-default stream should no longer see default stream synchronizations after memory transfers.
- Added a new constructor to the NVECTOR\_CUDA module that allows a user to provide custom allocate and free functions for the vector data array and internal reduction buffer. See §7.9.1 for more details.
- Added new Fortran 2003 interfaces for most NVECTOR modules. See Chapter 7 for more details on how to use the interfaces.
- Added three new NVECTOR utility functions, FN\_VGetVecAtIndexVectorArray, FN\_VSetVecAtIndexVectorArray, and FN\_VNewVectorArray, for working with N\_Vector arrays when using the Fortran 2003 interfaces. See §7.1.6 for more details.

#### SUNMatrix module changes

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

#### SUNLinearSolver module changes

• A new function was added to aid in creating custom SUNLINSOL objects. The constructor SUNLinSolNewEmpty allocates an "empty" generic SUNLINSOL with the object's content pointer and the function pointers in the operations structure initialized to NULL. When used in the constructor for custom objects this function will ease the introduction of any new optional

operations to the SUNLINSOL API by ensuring only required operations need to be set. See §9.3 for more details.

- The return type of the SUNLINSOL API function SUNLinSolLastFlag has changed from long int to sunindextype to be consistent with the type used to store row indices in dense and banded linear solver modules.
- Added a new optional operation to the SUNLINSOL API, SUNLinSolGetID, that returns a SUNLinearSolver\_ID for identifying the linear solver module.
- The SUNLINSOL API has been updated to make the initialize and setup functions optional.
- A new Sunlinsol (and Sunmatrix) implementation was added to facilitate the use of the SuperLU\_DIST library with Sundials. See §9.10 for more details.
- Added a new Sunlinsol implementation, SunlinearSolver\_cuSolverSp\_batchQR, which leverages the NVIDIA cuSOLVER sparse batched QR method for efficiently solving block diagonal linear systems on NVIDIA GPUs. See §9.12 for more details.
- Added three new accessor functions to the SUNLINSOL\_KLU module, SUNLinSol\_KLUGetSymbolic, SUNLinSol\_KLUGetNumeric, and SUNLinSol\_KLUGetCommon, to provide user access to the underlying KLU solver structures. See §9.9.2 for more details.
- Added new Fortran 2003 interfaces for most SUNLINSOL modules. See Chapter 9 for more details
  on how to use the interfaces.

#### SUNNonlinearSolver module changes

- A new function was added to aid in creating custom SUNNONLINSOL objects. The constructor SUNNonlinSolNewEmpty allocates an "empty" generic SUNNONLINSOL 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 SUNNONLINSOL API by ensuring only required operations need to be set. See §10.1.8 for more details.
- To facilitate the use of user supplied nonlinear solver convergence test functions the SUNNonlinSolSetConvTestFn function in the SUNNONLINSOL 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 SUNNonlinSolve function in the SUNNONLINSOL 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 SUNNONLINSOL API have been updated to remove unused input parameters. For more information on the nonlinear system formulation see §10.2 and for more details on the API functions see Chapter 10.
- Added a new sunnonlinsol implementation, sunnonlinsol\_petscsnes, which interfaces to the PETSc SNES nonlinear solver API. See §10.4 for more details.
- Added new Fortran 2003 interfaces for most Sunnonlinsol modules. See Chapter 10 for more details on how to use the interfaces.

#### **IDA** changes

- A bug was fixed in the IDA 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.

- 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, IDA 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 usersupplied Jacobian evaluation function when the attached linear solver has type SUNLINEARSOLVER\_DIRECT.
- Added the new functions, IDAGetCurentCj, IDAGetCurrentY, IDAGetCurrentYp, IDAComputeCurrentY, and IDAComputeCurrentYp which may be useful to users who choose to provide their own nonlinear solver implementations.
- Added a Fortran 2003 interface to IDA. See Chapter 5 for more details.

#### Changes in v4.1.0

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

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 NVECTOR is enabled).

The implementation header file ida\_impl.h is no longer installed. This means users who are directly manipulating the IDAMem structure will need to update their code to use IDA's public API.

Python is no longer required to run make test and make test\_install.

#### Changes in v4.0.2

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

Moved definitions of DLS and SPILS backwards compatibility functions to a source file. The symbols are now included in the IDA library, libsundials\_ida.

#### Changes in v4.0.1

No changes were made in this release.

#### Changes in v4.0.0

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 SUNLINSOL 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 SUNLINSOL implementations are included in Chapter 9. All IDA 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 SUNLINSOL implementations have been updated to follow the naming convention SUNLinSol\_\* where \* is the name of the linear solver. The new names are SUNLinSol\_Band, SUNLinSol\_Dense, SUNLinSol\_KLU, SUNLinSol\_LapackBand, SUNLinSol\_LapackDense, SUNLinSol\_SPEGGS, SUNLinSol\_SPEGMR, SUNLinSol\_SPEGMR,

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 IDA 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 SUNNONLINSOL API. This API will ease the addition of new nonlinear solver options and allow for external or user-supplied nonlinear solvers. The SUNNONLINSOL API and SUNDIALS provided modules are described in Chapter 10 and follow the same object oriented design and implementation used by the NVECTOR, SUNMATRIX, and SUNLINSOL modules. Currently two SUNNONLINSOL 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 IDA example programs have been updated to use generic SUNNONLINSOL modules.

By default IDA uses the SUNNONLINSOL\_NEWTON module. Since IDA 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 SUNNONLINSOL functions as needed. While SUNDIALS includes a fixed-point nonlinear solver module, it is not currently supported in IDA. For details on attaching a user-supplied nonlinear solver to IDA see Chapter 4. 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 IDA uses the SUNNONLINSOL\_NEWTON module by default).

Three fused vector operations and seven vector array operations have been added to the NVECTOR API. These *optional* operations are disabled by default and may be activated by calling vector specific routines after creating an NVECTOR (see Chapter 7 for more details). The new operations are intended to increase data reuse in vector operations, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. The fused operations are N\_VLinearCombination, N\_VScaleAddMulti, and N\_VDotProdMulti and the vector array operations are N\_VLinearCombinationVectorArray, N\_VScaleVectorArray, N\_VConstVectorArray, N\_VWrmsNormVectorArray, N\_VWrmsNormMaskVectorArray, N\_VScaleAddMultiVectorArray, and N\_VLinearCombinationVectorArray. If an NVECTOR implementation defines any of these operations as NULL, then standard NVECTOR operations will automatically be called as necessary to complete the computation.

Multiple updates to NVECTOR\_CUDA were made:

- Changed N\_VGetLength\_Cuda to return the global vector length instead of the local vector length.
- Added N\_VGetLocalLength\_Cuda to return the local vector length.
- Added N\_VGetMPIComm\_Cuda to return the MPI communicator used.
- Removed the accessor functions in the namespace suncudavec.
- Changed the N\_VMake\_Cuda function to take a host data pointer and a device data pointer instead
  of an N\_VectorContent\_Cuda object.
- Added the ability to set the cudaStream\_t used for execution of the NVECTOR\_CUDA kernels. See the function N\_VSetCudaStreams\_Cuda.
- Added N\_VNewManaged\_Cuda, N\_VMakeManaged\_Cuda, and N\_VIsManagedMemory\_Cuda functions
  to accommodate using managed memory with the NVECTOR\_CUDA.

Multiple changes to NVECTOR\_RAJA were made:

- Changed N\_VGetLength\_Raja to return the global vector length instead of the local vector length.
- Added N\_VGetLocalLength\_Raja to return the local vector length.
- Added N\_VGetMPIComm\_Raja to return the MPI communicator used.
- Removed the accessor functions in the namespace suncudavec.

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

#### Changes in v3.2.1

The changes in this minor release include the following:

- Fixed a bug in the CUDA NVECTOR where the N\_VInvTest operation could write beyond the allocated vector data.
- Fixed library installation path for multiarch systems. This fix changes the default library installation path to CMAKE\_INSTALL\_PREFIX/CMAKE\_INSTALL\_LIBDIR from CMAKE\_INSTALL\_PREFIX/lib. CMAKE\_INSTALL\_LIBDIR is automatically set, but is available as a CMake option that can modified.

#### Changes in v3.2.0

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

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

Changed the name of the RAJA NVECTOR library to libsundials\_nveccudaraja.lib from libsundials\_nvecraja.lib to better reflect that we only support CUDA as a backend for RAJA currently.

Several changes were made to the build system:

- CMake 3.1.3 is now the minimum required CMake version.
- Deprecate the behavior of the SUNDIALS\_INDEX\_TYPE CMake option and added the SUNDIALS\_INDEX\_SIZE CMake option to select the sunindextype integer size.
- The native CMake FindMPI module is now used to locate an MPI installation.
- If MPI is enabled and MPI compiler wrappers are not set, the build system will check if CMAKE\_<language>\_COMPILER can compile MPI programs before trying to locate and use an MPI installation.
- The previous options for setting MPI compiler wrappers and the executable for running MPI programs have been 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.

#### Changes in v3.1.2

The changes in this minor release include the following:

- Updated the minimum required version of CMake to 2.8.12 and enabled using rpath by default to locate shared libraries on OSX.
- Fixed Windows specific problem where sunindextype was not correctly defined when using 64-bit integers for the SUNDIALS index type. On Windows sunindextype is now defined as the MSVC basic type \_\_int64.
- Added sparse SUNMatrix "Reallocate" routine to allow specification of the nonzero storage.
- Updated the KLU SUNLinearSolver module to set constants for the two reinitialization types, and fixed a bug in the full reinitialization approach where the sparse SUNMatrix pointer would go out of scope on some architectures.
- Updated the "ScaleAdd" and "ScaleAddl" implementations in the sparse SUNMatrix module to more optimally handle the case where the target matrix contained sufficient storage for the sum, but had the wrong sparsity pattern. The sum now occurs in-place, by performing the sum backwards in the existing storage. However, it is still more efficient if the user-supplied Jacobian routine allocates storage for the sum  $I + \gamma J$  manually (with zero entries if needed).
- Changed the LICENSE install path to instdir/include/sundials.

#### Changes in v3.1.1

The changes in this minor release include the following:

- Fixed a potential memory leak in the SPGMR and SPFGMR linear solvers: if "Initialize" was called multiple times then the solver memory was reallocated (without being freed).
- Updated KLU SUNLINSOL 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 NVECTOR and SUNMATRIX header files.
- Added missing prototype for IDASpilsGetNumJTSetupEvals.
- Fixed an indexing bug in the CUDA NVECTOR implementation of N\_VWrmsNormMask and revised the RAJA NVECTOR implementation of N\_VWrmsNormMask to work with mask arrays using values other than zero or one. Replaced double with realtype in the RAJA vector test functions.
- Fixed compilation issue with GCC 7.3.0 and Fortran programs that do not require a SUNMATRIX 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.

#### Changes in v3.1.0

Added NVECTOR print functions that write vector data to a specified file (e.g., N\_VPrintFile\_Serial).

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

#### Changes in v3.0.0

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

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

All indices for data structures were updated to a new sunindextype that can be configured to be a 32- or 64-bit integer data index type. sunindextype is defined to be int32\_t or int64\_t when portable types are supported, otherwise it is defined as int or long int. The Fortran interfaces continue to use long int for indices, except for their sparse matrix interface that now uses the new sunindextype. This new flexible capability for index types includes interfaces to PETSc, hypre, SuperLU\_MT, and KLU with either 32-bit or 64-bit capabilities depending how the user configures SUNDIALS.

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

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

The file include/sundials\_fconfig.h was added. This file contains SUNDIALS type information for use in Fortran programs.

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

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

In addition, numerous changes were made to the build system. These include the addition of separate BLAS\_ENABLE and BLAS\_LIBRARIES CMake variables, additional error checking during CMake configuration, minor bug fixes, and renaming CMake options to enable/disable examples for greater clarity and an added option to enable/disable Fortran 77 examples. These changes included changing EXAMPLES\_ENABLE to EXAMPLES\_ENABLE\_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 ida.h.

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

#### Changes in v2.9.0

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

Each NVECTOR module now includes a function, N\_VGetVectorID, that returns the NVECTOR module name.

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 limit function. This ensures that these solver counters are initialized upon linear solver instantiation as well as at the beginning of the problem solution.

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

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 IDA solver, to the Fortran interfaces, to the examples, to installation-related files, and to the user documentation.

#### Changes in v2.8.0

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

Otherwise, only relatively minor modifications were made to IDA:

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.

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 NVECTOR modules have been added for thread-parallel computing environments — one for OpenMP, denoted NVECTOR\_OPENMP, and one for Pthreads, denoted NVECTOR\_PTHREADS.

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

#### Changes in v2.7.0

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

A large number of minor errors have been fixed. Among these are the following: After the solver memory is created, it is set to zero before being filled. To be consistent with IDAS, IDA uses the function IDAGetDky for optional output retrieval. 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/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\_LIB is either defined (with no value) or not defined.

#### Changes in v2.6.0

Two new features were added in this release: (a) a new linear solver module, based on BLAS and LAPACK for both dense and banded matrices, and (b) option to specify which direction of zero-crossing is to be monitored while performing rootfinding.

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

#### Changes in v2.5.0

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

A bug was fixed in the internal difference-quotient dense and banded Jacobian approximations, related to the estimation of the perturbation (which could have led to a failure of the linear solver when zero components with sufficiently small absolute tolerances were present).

The user interface to the consistent initial conditions calculations was modified. The IDACalcIC arguments t0, yy0, and yp0 were removed and a new function, IDAGetconsistentIC is provided (see

 $\S4.5.5$  and  $\S4.5.10.3$  for details).

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

#### Changes in v2.4.0

FIDA, a FORTRAN-C interface module, was added (for details see Chapter 5.2).

IDASPBCG and IDASPTFQMR modules have been added to interface with the Scaled Preconditioned Bi-CGstab (SPBCGS) and Scaled Preconditioned Transpose-Free Quasi-Minimal Residual (SPTFQMR) linear solver modules, respectively (for details see Chapter 4). At the same time, function type names for Scaled Preconditioned Iterative Linear Solvers were added for the user-supplied Jacobian-times-vector and preconditioner setup and solve functions.

The rootfinding feature was added, whereby the roots of a set of given functions may be computed during the integration of the DAE system.

A user-callable routine was added to access the estimated local error vector.

The deallocation functions now take as arguments the address of the respective memory block pointer.

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

#### Changes in v2.3.0

The user interface has been further refined. Several functions used for setting optional inputs were combined into a single one. An optional user-supplied routine for setting the error weight vector was added. Additionally, to resolve potential variable scope issues, all SUNDIALS solvers release user data right after its use. The build systems has been further improved to make it more robust.

#### Changes in v2.2.2

Minor corrections and improvements were made to the build system. A new chapter in the User Guide was added — with constants that appear in the user interface.

#### Changes in v2.2.1

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

#### Changes in v2.2.0

The major changes from the previous version involve a redesign of the user interface across the entire SUNDIALS suite. We have eliminated the mechanism of providing optional inputs and extracting optional statistics from the solver through the iopt and ropt arrays. Instead, IDA now provides a set of routines (with prefix IDASet) to change the default values for various quantities controlling the solver and a set of extraction routines (with prefix IDAGet) to extract statistics after return from the main solver routine. Similarly, each linear solver module provides its own set of Set- and Get-type routines. For more details see §4.5.8 and §4.5.10.

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

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

#### 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 IDA 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 (§3.1) and the software organization of the IDA solver (§3.2).
- Chapter 4 is the main usage document for IDA for C applications. It includes a complete description of the user interface for the integration of DAE initial value problems.
- In Chapter 5.2, we describe FIDA, an interface module for the use of IDA with FORTRAN applications.
- Chapter 7 gives a brief overview of the generic NVECTOR module shared among the various components of SUNDIALS, as well as details on the NVECTOR implementations provided with SUNDIALS.
- Chapter 8 gives a brief overview of the generic SUNMATRIX module shared among the various components of SUNDIALS, and details on the SUNMATRIX implementations provided with SUNDIALS: a dense implementation (§8.3), a banded implementation (§8.4) and a sparse implementation (§8.5).
- Chapter 9 gives a brief overview of the generic SUNLINSOL module shared among the various components of SUNDIALS. This chapter contains details on the SUNLINSOL implementations provided with SUNDIALS. The chapter also contains details on the SUNLINSOL implementations provided with SUNDIALS that interface with external linear solver libraries.
- Chapter 10 describes the SUNNONLINSOL API and nonlinear solver implementations shared among the various components of SUNDIALS.
- Finally, in the appendices, we provide detailed instructions for the installation of IDA, within the structure of SUNDIALS (Appendix A), as well as a list of all the constants used for input to and output from IDA functions (Appendix B).

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



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

#### 1.3 SUNDIALS Release License

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



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

#### 1.3.1 BSD 3-Clause License

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

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

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

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

#### 1.3.2 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.

This work was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights.

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

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

#### 1.3.3 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**

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

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

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

#### 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, IDA provides a routine that computes consistent initial conditions from a user's initial guess [15]. 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 IDA 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, IDA 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 IDA is the variable-order, variable-coefficient BDF (Backward Differentiation Formula), in fixed-leading-coefficient form [11]. The method order ranges from 1 to 5, with the BDF of order q given by the multistep formula

$$\sum_{i=0}^{q} \alpha_{n,i} y_{n-i} = h_n \dot{y}_n \,, \tag{2.2}$$

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 (2.2) to the DAE system (2.1) results in a nonlinear algebraic system to be solved at each step:

$$G(y_n) \equiv F\left(t_n, y_n, h_n^{-1} \sum_{i=0}^{q} \alpha_{n,i} y_{n-i}\right) = 0.$$
 (2.3)

By default IDA solves (2.3) with a Newton iteration but IDA also allows for user-defined nonlinear solvers (see Chapter 10). Each Newton iteration requires the solution of a linear system of the form

$$J[y_{n(m+1)} - y_{n(m)}] = -G(y_{n(m)}), (2.4)$$

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

$$J = \frac{\partial G}{\partial y} = \frac{\partial F}{\partial y} + \alpha \frac{\partial F}{\partial \dot{y}}, \qquad (2.5)$$

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

For the solution of the linear systems within the Newton iteration, IDA provides several choices, including the option of a user-supplied linear solver module (see Chapter 9). The linear solver modules distributed with SUNDIALS are organized in two families, a *direct* family comprising direct linear solvers for dense, banded, or sparse matrices and a *spils* family comprising scaled preconditioned iterative (Krylov) linear solvers. The methods offered through these modules are as follows:

- dense direct solvers, using either an internal implementation or a BLAS/LAPACK implementation (serial or threaded vector modules only),
- band direct solvers, using either an internal implementation or a BLAS/LAPACK implementation (serial or threaded vector modules only),
- sparse direct solver interfaces, using either the KLU sparse solver library [22, 3], or the threadenabled SuperLU\_MT sparse solver library [39, 24, 9] (serial or threaded vector modules only) [Note that users will need to download and install the KLU or SUPERLUMT packages independent of IDA],
- SPGMR, a scaled preconditioned GMRES (Generalized Minimal Residual method) solver with or without restarts,
- SPFGMR, a scaled preconditioned FGMRES (Flexible Generalized Minimal Residual method) solver with or without restarts,
- SPBCGS, a scaled preconditioned Bi-CGStab (Bi-Conjugate Gradient Stable method) solver,
- SPTFQMR, a scaled preconditioned TFQMR (Transpose-Free Quasi-Minimal Residual method) solver, or
- PCG, a scaled preconditioned CG (Conjugate Gradient method) solver.

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 [13]. For the *spils* linear solvers with IDA, 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, IDA uses a weighted root-mean-square norm, denoted  $\|\cdot\|_{WRMS}$ , 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

$$W_i = 1/[\text{RTOL} \cdot |y_i| + \text{ATOL}_i]. \tag{2.6}$$

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  $\alpha$  in J. However, in the case that a matrix-free iterative linear solver is

2.1 IVP solution 21

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
- $\bullet$  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 IDA 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

$$S\|\delta_m\| < 0.33\,,\tag{2.7}$$

where S = R/(R-1) whenever m > 1 and  $R \le 0.9$ . 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  $\alpha \ne \bar{\alpha}$ . Note that at m = 1, the convergence test (2.7) 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  $\delta_1$  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 dense or band Sunmatrix objects, the Jacobian J defined in (2.5) can be either supplied by the user or have IDA compute one internally by difference quotients. In the latter case, we use the approximation

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

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 (2.6). We note that with sparse and user-supplied SUNMATRIX 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  $\sigma$ .

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

$$\|LTE\|_{WRMS} \le 1$$
.

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

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

and so the norm of LTE is estimated as  $|C| \cdot ||\Delta_n||$ . In addition, IDA 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 IDA is

$$\max\{|C|, \bar{C}\} \|\Delta_n\| \le 1. \tag{2.8}$$

A user option is available by which the algebraic components of the error vector are omitted from the test (2.8), if these have been so identified.

In IDA, 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 (2.8) 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, IDA 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

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

where  $\phi(k)$  is a modified divided difference of order k that is retained by IDA (and behaves asymptotically as  $h^k$ ). Thus the local truncation errors are estimated as  $\mathrm{ELTE}(q') = C(q') \|\phi(q'+1)\|$  to select step sizes. But the choice of order in IDA 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  $T(1) \le T(2)/2$ , or (b) q > 2 and  $\max\{T(q-1), T(q-2)\} \le T(q)$ ; otherwise q' = q. Next the local error test (2.8) 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 ELTE(q), and, with safety factors, is given by

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

The value of  $\eta$  is adjusted so that  $0.25 \le \eta \le 0.9$  before setting  $h \leftarrow h' = \eta h$ . If the local error test fails a second time, IDA uses  $\eta = 0.25$ , and on the third and subsequent failures it uses q = 1 and  $\eta = 0.25$ . After 10 failures, IDA 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, IDA 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  $T(q 1) \le \min\{T(q), T(q + 1)\}$ ;
- else reset  $q \leftarrow q + 1$  if T(q + 1) < T(q);

• leave q unchanged otherwise [then  $T(q-1) > T(q) \le T(q+1)$ ].

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

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

The value of  $\eta$  is adjusted such that (a) if  $\eta > 2$ ,  $\eta$  is reset to 2; (b) if  $\eta \le 1$ ,  $\eta$  is restricted to  $0.5 \le \eta \le 0.9$ ; 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 [11] for details.

IDA 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, IDA 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 §4.5.2 to satisfy the inequality constraints.

Normally, IDA takes steps until a user-defined output value  $t = t_{\text{out}}$  is overtaken, and then computes  $y(t_{\text{out}})$  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 IDA not to integrate past a given stopping point  $t = t_{\text{stop}}$ .

#### 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), IDA 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 IDA, 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 [13] for an extensive study of preconditioners for reaction-transport systems).

Typical preconditioners used with IDA are based on approximations to the iteration matrix of the systems involved; in other words,  $P \approx \frac{\partial F}{\partial y} + \alpha \frac{\partial F}{\partial y}$ , where  $\alpha$  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 IDA solver has been augmented to include a rootfinding feature. This means that, while integrating the Initial Value Problem (2.1), IDA 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 IDA. 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 [30]. In addition, each time g is computed, IDA 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, IDA computes g at  $t + \delta$  for a small increment  $\delta$ , slightly further in the direction of integration, and if any  $g_i(t + \delta) = 0$  also, IDA stops and reports an error. This way, each time IDA 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, IDA 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

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

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  $\alpha$  a weight parameter. On the first two passes through the loop,  $\alpha$  is set to 1, making  $t_{mid}$  the secant method value. Thereafter,  $\alpha$  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,  $\alpha$  is set to 1. If the two sides were the same,  $\alpha$  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 .1 and .5 (.5 being the midpoint), and the actual distance from the endpoint is at least  $\tau/2$ .

# Chapter 3

# **Code Organization**

# 3.1 SUNDIALS organization

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

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



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

26 Code Organization

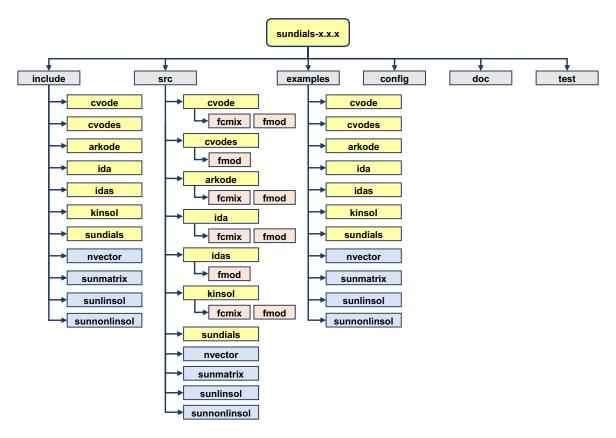


Figure 3.2: Directory structure of the SUNDIALS source tree.

following is a list of the solver packages presently available, and the basic functionality of each:

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

Note for modules that provide interfaces to third-party libraries (i.e., LAPACK, KLU, SUPERLUMT, SuperLU\_DIST, hypre, Petsc, Trilinos, and Raja) users will need to download and compile those packages independently.

# 3.2 IDA organization

The IDA package is written in the ANSI C language. 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 IDA package is shown in Figure 3.3. The central integration module, implemented in the files ida.h, ida\_impl.h, and ida.c, deals with the evaluation of integration

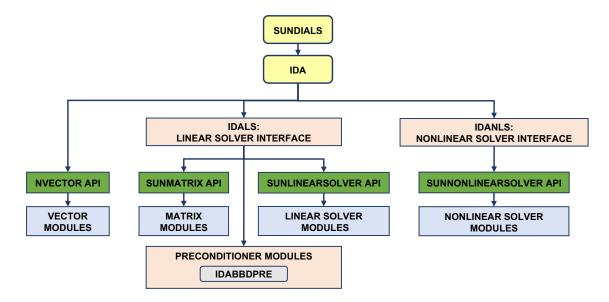


Figure 3.3: Overall structure diagram of the IDA package. Modules specific to IDA begin with "IDA" (IDALS, IDANLS, and IDABBDPRE), all other items correspond to generic SUNDIALS vector, matrix, and solver modules (see Figure 3.1).

coefficients, estimation of local error, selection of stepsize and order, and interpolation to user output points, among other issues.

IDA utilizes generic linear and nonlinear solver modules defined by the SUNLINSOL API (see Chapter 9) and SUNNONLINSOL API (see Chapter 10) respectively. As such, IDA has no knowledge of the method being used to solve the linear and nonlinear systems that arise in each time step. For any given user problem, there exists a single nonlinear solver interface and, if necessary, a linear system solver interface is specified, and invoked as needed during the integration. While SUNDIALS includes a fixed-point nonlinear solver module, it is not currently supported in IDA (note the fixed-point module is listed in Figure 3.1 but not Figure 3.3).

IDA now has a single unified linear solver interface, IDALS, supporting both direct and iterative linear solvers built using the generic SUNLINSOL API (see Chapter 9). These solvers may utilize a SUNMATRIX object (see Chapter 8) for storing Jacobian information, or they may be matrix-free. Since IDA can operate on any valid SUNLINSOL implementation, the set of linear solver modules available to IDA will expand as new SUNLINSOL modules are developed.

For users employing dense or banded Jacobian matrices, IDALS includes algorithms for their approximation through difference quotients, but the user also has the option of supplying the Jacobian (or an approximation to it) directly. This user-supplied routine is required when using sparse or user-supplied Jacobian matrices.

For users employing matrix-free iterative linear solvers, IDALS includes an algorithm for the approximation by difference quotients of the product between the Jacobian matrix and a vector, Jv. Again, the user has the option of providing routines for this operation, in two phases: setup (preprocessing of Jacobian data) and multiplication.

For preconditioned iterative methods, the preconditioning must be supplied by the user, again in two phases: setup and solve. While there is no default choice of preconditioner analogous to the difference-quotient approximation in the direct case, the references [13, 17], together with the example and demonstration programs included with IDA, offer considerable assistance in building preconditioners.

IDA's linear solver interface consists of four primary routines, devoted to (1) memory allocation and initialization, (2) setup of the matrix data involved, (3) solution of the system, and (4) freeing of memory. The setup and solution phases are separate because the evaluation of Jacobians and preconditioners is done only periodically during the integration, as required to achieve convergence.

28 Code Organization

The call list within the central IDA module to each of the four associated functions is fixed, thus allowing the central module to be completely independent of the linear system method.

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

All state information used by IDA to solve a given problem is saved in a structure, and a pointer to that structure is returned to the user. There is no global data in the IDA package, and so, in this respect, it is reentrant. State information specific to the linear solver is saved in a separate structure, a pointer to which resides in the IDA memory structure. The reentrancy of IDA was motivated by the situation where two or more problems are solved by intermixed calls to the package from one user program.

# Chapter 4

# Using IDA for C Applications

This chapter is concerned with the use of IDA for the integration of DAEs in a C language setting. The following sections treat the header files, the layout of the user's main program, description of the IDA user-callable functions, and description of user-supplied functions.

The sample programs described in the companion document [34] may also be helpful. Those codes may be used as templates (with the removal of some lines involved in testing), and are included in the IDA package.

Users with applications written in FORTRAN should see Chapter 5.2, which describes the FORTRAN/C interface module.

The user should be aware that not all SUNLINSOL and SUNMATRIX modules are compatible with all NVECTOR implementations. Details on compatibility are given in the documentation for each SUNMATRIX module (Chapter 8) and each SUNLINSOL module (Chapter 9). For example, NVECTOR\_PARALLEL is not compatible with the dense, banded, or sparse SUNMATRIX types, or with the corresponding dense, banded, or sparse SUNLINSOL modules. Please check Chapters 8 and 9 to verify compatibility between these modules. In addition to that documentation, we note that the preconditioner module IDABBDPRE can only be used with NVECTOR\_PARALLEL. It is not recommended to use a threaded vector module with SuperLU\_MT unless it is the NVECTOR\_OPENMP module, and SuperLU\_MT is also compiled with OpenMP.

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

# 4.1 Access to library and header files

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

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

- *libdir*/libsundials\_ida. *lib*,
- libdir/libsundials\_nvec\*.lib,

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

- incdir/include/ida
- incdir/include/sundials
- incdir/include/nvector

- incdir/include/sunmatrix
- incdir/include/sunlinsol
- incdir/include/sunnonlinsol

The directories *libdir* and *incdir* are the install library and include directories, respectively. For a default installation, these are *instdir*/lib and *instdir*/include, respectively, where *instdir* is the directory where SUNDIALS was installed (see Appendix A).

Note that an application cannot link to both the IDA and IDAS 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.

# 4.2 Data types

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

# 4.2.1 Floating point types

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

Additionally, based on the current precision, sundials\_types.h defines BIG\_REAL to be the largest value representable as a realtype, SMALL\_REAL to be the smallest value representable as a realtype, and UNIT\_ROUNDOFF to be the difference between 1.0 and the minimum realtype greater than 1.0.

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

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

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

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

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

4.3 Header files 31

# 4.2.2 Integer types used for indexing

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

A user program which uses sunindextype to handle indices will work with both index storage types except for any calls to index storage-specific external libraries. Our C and C++ example programs use sunindextype. Users can, however, use any compatible type (e.g., int, long int, int32\_t, int64\_t, or long long int) in their code, assuming that this usage is consistent with the typedef for sunindextype on their architecture. Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use sunindextype, so long as the SUNDIALS libraries use the appropriate index storage type (for details see §A.1.2).

# 4.3 Header files

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

• ida/ida.h, the header file for IDA, which defines the several types and various constants, and includes function prototypes. This includes the header file for IDALS, ida/ida\_ls.h.

Note that ida.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 NVECTOR implementation header file, of the form nvector/nvector\_\*\*\*.h. See Chapter 7 for the appropriate name. This file in turn includes the header file sundials\_nvector.h which defines the abstract N\_Vector data type.

If using a non-default nonlinear solver module, or when interacting with a SUNNONLINSOL module directly, the calling program must also include a SUNNONLINSOL implementation header file, of the form sunnonlinsol\_\*\*\*.h where \*\*\* is the name of the nonlinear solver module (see Chapter 10 for more information). This file in turn includes the header file sundials\_nonlinearsolver.h which defines the abstract SUNNonlinearSolver 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), a linear solver module header file is also required. The header files corresponding to the various Sundials-provided linear solver modules available for use with IDA are:

#### • Direct linear solvers:

- sunlinsol/sunlinsol\_dense.h, which is used with the dense linear solver module, SUN-LINSOL\_DENSE;
- sunlinsol/sunlinsol\_band.h, which is used with the banded linear solver module, SUN-LINSOL\_BAND;
- sunlinsol/sunlinsol\_lapackdense.h, which is used with the LAPACK dense linear solver module, SUNLINSOL\_LAPACKDENSE;
- sunlinsol/sunlinsol\_lapackband.h, which is used with the LAPACK banded linear solver module. SUNLINSOL\_LAPACKBAND:
- sunlinsol/sunlinsol\_klu.h, which is used with the KLU sparse linear solver module, SUNLINSOL KLU:
- sunlinsol/sunlinsol\_superlumt.h, which is used with the SUPERLUMT sparse linear solver module, SUNLINSOL\_SUPERLUMT;

#### • Iterative linear solvers:

- sunlinsol/sunlinsol\_spgmr.h, which is used with the scaled, preconditioned GMRES Krylov linear solver module, SUNLINSOL\_SPGMR;
- sunlinsol/sunlinsol\_spfgmr.h, which is used with the scaled, preconditioned FGMRES
   Krylov linear solver module, SUNLINSOL\_SPFGMR;
- sunlinsol/sunlinsol\_spbcgs.h, which is used with the scaled, preconditioned Bi-CGStab Krylov linear solver module, SUNLINSOL\_SPBCGS;
- sunlinsol/sunlinsol\_sptfqmr.h, which is used with the scaled, preconditioned TFQMR
   Krylov linear solver module, SUNLINSOL\_SPTFQMR;
- sunlinsol/sunlinsol\_pcg.h, which is used with the scaled, preconditioned CG Krylov linear solver module, SUNLINSOL\_PCG;

The header files for the SUNLINSOL\_DENSE and SUNLINSOL\_LAPACKDENSE linear solver modules include the file sunmatrix/sunmatrix\_dense.h, which defines the SUNMATRIX\_DENSE matrix module, as as well as various functions and macros acting on such matrices.

The header files for the SUNLINSOL\_BAND and SUNLINSOL\_LAPACKBAND linear solver modules include the file sunmatrix/sunmatrix\_band.h, which defines the SUNMATRIX\_BAND matrix module, as as well as various functions and macros acting on such matrices.

The header files for the SUNLINSOL\_KLU and SUNLINSOL\_SUPERLUMT sparse linear solvers include the file sunmatrix\_sparse.h, which defines the SUNMATRIX\_SPARSE matrix module, as well as various functions and macros acting on such matrices.

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

Other headers may be needed, according to the choice of preconditioner, etc. For example, in the idaFoodWeb\_kry\_p example (see [34]), preconditioning is done with a block-diagonal matrix. For this, even though the SUNLINSOL\_SPGMR linear solver is used, the header sundials/sundials\_dense.h is included for access to the underlying generic dense matrix arithmetic routines.

# 4.4 A skeleton of the user's main program

The following is a skeleton of the user's main program (or calling program) for the integration of a DAE IVP. Most of the steps are independent of the NVECTOR, SUNMATRIX, SUNLINSOL, and SUNNONLINSOL implementations used. For the steps that are not, refer to Chapter 7, 8, 9, and 10 for the specific name of the function to be called or macro to be referenced.

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

For example, call MPI\_Init to initialize MPI if used, or set num\_threads, the number of threads to use within the threaded vector functions, if used.

# 2. Set problem dimensions etc.

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

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

#### 3. Set vectors of initial values

To set the vectors y0 and yp0 to initial values for y and  $\dot{y}$ , use the appropriate functions defined by the particular NVECTOR implementation.

For native SUNDIALS vector implementations, use a call of the form  $y0 = N_VMake_***(..., ydata)$  if the realtype array ydata containing the initial values of y already exists. Otherwise, create a new vector by making a call of the form  $y0 = N_VNew_***(...)$ , and then set its elements

by accessing the underlying data with a call of the form ydata = N\_VGetArrayPointer(y0). See §7.3-7.6 for details.

For the hypre and PETSc vector wrappers, first create and initialize the underlying vector and then create an NVECTOR wrapper with a call of the form y0 = N\_VMake\_\*\*\*(yvec), where yvec is a hypre or PETSc vector. Note that calls like N\_VNew\_\*\*\*(...) and N\_VGetArrayPointer(...) are not available for these vector wrappers. See §7.7 and §7.8 for details.

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

#### 4. Create IDA object

Call ida\_mem = IDACreate() to create the IDA memory block. IDACreate returns a pointer to the IDA memory structure. See §4.5.1 for details. This void \* pointer must then be passed as the first argument to all subsequent IDA function calls.

#### 5. Initialize IDA solver

Call IDAInit(...) to provide required problem specifications (residual function, initial time, and initial conditions), allocate internal memory for IDA, and initialize IDA. IDAInit returns an error flag to indicate success or an illegal argument value. See §4.5.1 for details.

#### 6. Specify integration tolerances

Call IDASStolerances(...) or IDASVtolerances(...) to specify, respectively, a scalar relative tolerance and scalar absolute tolerance, or a scalar relative tolerance and a vector of absolute tolerances. Alternatively, call IDAWFtolerances to specify a function which sets directly the weights used in evaluating WRMS vector norms. See §4.5.2 for details.

#### 7. Create matrix object

If a nonlinear solver requiring a linear solver will be used (e.g., the default Newton iteration) and the linear solver will be a matrix-based linear solver, then a template Jacobian matrix must be created by calling the appropriate constructor function defined by the particular SUNMATRIX implementation.

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

```
SUNMatrix J = SUNBandMatrix(...);
or
SUNMatrix J = SUNDenseMatrix(...);
or
SUNMatrix J = SUNSparseMatrix(...);
```

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

#### 8. Create linear solver object

If a nonlinear solver requiring a linear solver is chosen (e.g., the default Newton iteration), then the desired linear solver object must be created by calling the appropriate constructor function defined by the particular SUNLINSOL implementation.

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

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

where \* can be replaced with "Dense", "SPGMR", or other options, as discussed in §4.5.3 and Chapter 9.

#### 9. Set linear solver optional inputs

Call \*Set\* functions from the selected linear solver module to change optional inputs specific to that linear solver. See the documentation for each SUNLINSOL module in Chapter 9 for details.

#### 10. Attach linear solver module

If a nonlinear solver requiring a linear solver is chosen (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 the following call (for details see §4.5.3):

```
ier = IDASetLinearSolver(...);
```

#### 11. Set optional inputs

Optionally, call IDASet\* functions to change from their default values any optional inputs that control the behavior of IDA. See §4.5.8.1 and §4.5.8 for details.

#### 12. Create nonlinear solver object (optional)

If using a non-default nonlinear solver (see §4.5.4), then create the desired nonlinear solver object by calling the appropriate constructor function defined by the particular SUNNONLINSOL implementation (e.g., NLS = SUNNonlinSol\_\*\*\*\*(...); where \*\*\* is the name of the nonlinear solver (see Chapter 10 for details).

# 13. Attach nonlinear solver module (optional)

If using a non-default nonlinear solver, then initialize the nonlinear solver interface by attaching the nonlinear solver object by calling ier = IDASetNonlinearSolver(ida\_mem, NLS); (see §4.5.4 for details).

#### 14. Set nonlinear solver optional inputs (optional)

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 IDAInit if using the default nonlinear solver or after attaching a new nonlinear solver to IDA, otherwise the optional inputs will be overridden by IDA defaults. See Chapter 10 for more information on optional inputs.

# 15. Correct initial values

Optionally, call IDACalcIC to correct the initial values y0 and yp0 passed to IDAInit. See §4.5.5. Also see §4.5.8.3 for relevant optional input calls.

#### 16. Specify rootfinding problem

Optionally, call IDARootInit to initialize a rootfinding problem to be solved during the integration of the DAE system. See §4.5.6 for details, and see §4.5.8.4 for relevant optional input calls.

### 17. Advance solution in time

For each point at which output is desired, call flag = 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 §4.5.7 for details.

#### 18. Get optional outputs

Call IDA\*Get\* functions to obtain optional output. See §4.5.10 for details.

#### 19. Deallocate memory for solution vectors

Upon completion of the integration, deallocate memory for the vectors yret and ypret (or y and yp) by calling the appropriate destructor function defined by the NVECTOR implementation:

N\_VDestroy(yret); and similarly for ypret.

#### 20. Free solver memory

IDAFree (&ida\_mem) to free the memory allocated for IDA.

#### 21. Free nonlinear solver memory (optional)

If a non-default nonlinear solver was used, then call SUNNonlinSolFree(NLS) to free any memory allocated for the SUNNONLINSOL object.

#### 22. Free linear solver and matrix memory

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

#### 23. Finalize MPI, if used

Call MPI\_Finalize() to terminate MPI.

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

Table 4.1: SUNDIA	ALS linear s	solver i	inter	faces	and '	vector	imple	emen	tatio	ns th	at car	ı be us	ed for	each.
					Ы	35								

Linear Solver	Serial	Parallel (MPI)	OpenMP	pThreads	hypre	PETSC	CUDA	RAJA	User Supp.
Dense	<b>√</b>		<b>√</b>	<b>√</b>					<b>√</b>
Band	<b>√</b>		<b>√</b>	<b>√</b>					<b>√</b>
LapackDense	<b>√</b>		<b>√</b>	<b>√</b>					<b>√</b>
LapackBand	<b>√</b>		<b>√</b>	<b>√</b>					<b>√</b>
KLU	<b>√</b>		<b>√</b>	<b>√</b>					$\checkmark$
SUPERLUMT	<b>√</b>		<b>√</b>	<b>√</b>					$\checkmark$
SPGMR	<b>√</b>	✓	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>	$\checkmark$
SPFGMR	<b>√</b>	✓	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>	$\checkmark$
SPBCGS	<b>√</b>	✓	<b>√</b>						
SPTFQMR	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>
PCG	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>
User Supp.	✓	✓	<b>√</b>	✓	<b>√</b>	<b>√</b>	✓	<b>√</b>	<b>√</b>

# 4.5 User-callable functions

This section describes the IDA functions that are called by the user to set up and solve a DAE. Some of these are required. However, starting with §4.5.8, the functions listed involve optional inputs/outputs or restarting, and those paragraphs can be skipped for a casual use of IDA. In any case, refer to §4.4 for the correct order of these calls.

On an error, each user-callable function returns a negative value and sends an error message to the error handler routine, which prints the message on stderr by default. However, the user can set a file as error output or can provide his own error handler function (see §4.5.8.1).

#### 4.5.1 IDA initialization and deallocation functions

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

#### IDACreate

Call ida\_mem = IDACreate();

Description The function IDACreate instantiates an IDA solver object.

Arguments IDACreate has no arguments.

Return value If successful, IDACreate returns a pointer to the newly created IDA memory block (of

type void \*). Otherwise it returns NULL.

F2003 Name FIDACreate

# IDAInit

Call flag = IDAInit(ida\_mem, res, t0, y0, yp0);

Description The function IDAInit provides required problem and solution specifications, allocates

internal memory, and initializes IDA.

Arguments ida\_mem (void \*) pointer to the IDA memory block returned by IDACreate.

res (IDAResFn) is the C function which computes the residual function F in the

DAE. This function has the form res(t, yy, yp, resval, user\_data). For

full details see  $\S4.6.1$ .

to (realtype) is the initial value of t.

y0 (N\_Vector) is the initial value of y.

yp0 (N\_Vector) is the initial value of  $\dot{y}$ .

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

IDA\_SUCCESS The call to IDAInit was successful.

IDA\_MEM\_NULL The IDA 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 IDAInit has an illegal value.

Notes If an error occurred, IDAInit also sends an error message to the error handler function.

F2003 Name FIDAInit

#### IDAFree

Call IDAFree(&ida\_mem);

Description The function IDAFree frees the pointer allocated by a previous call to IDACreate.

Arguments The argument is the pointer to the IDA memory block (of type void \*).

Return value The function IDAFree has no return value.

F2003 Name FIDAFree

# 4.5.2 IDA 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.

#### IDASStolerances

Call flag = IDASStolerances(ida\_mem, reltol, abstol);

Description The function IDASStolerances specifies scalar relative and absolute tolerances.

Arguments ida\_mem (void \*) pointer to the IDA memory block returned by IDACreate.

reltol (realtype) is the scalar relative error tolerance. abstol (realtype) is the scalar absolute error tolerance.

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

IDA\_SUCCESS The call to IDASStolerances was successful.

IDA\_MEM\_NULL The IDA memory block was not initialized through a previous call to IDACreate.

IDA\_NO\_MALLOC The allocation function IDAInit has not been called.

IDA\_ILL\_INPUT One of the input tolerances was negative.

F2003 Name FIDASStolerances

#### IDASVtolerances

Call flag = IDASVtolerances(ida\_mem, reltol, abstol);

Description The function IDASVtolerances specifies scalar relative tolerance and vector absolute

tolerances.

Arguments ida\_mem (void \*) pointer to the IDA memory block returned by IDACreate.

reltol (realtype) is the scalar relative error tolerance.

abstol (N\_Vector) is the vector of absolute error tolerances.

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

IDA\_SUCCESS The call to IDASVtolerances was successful.

IDA\_MEM\_NULL The IDA memory block was not initialized through a previous call to

IDACreate

IDA\_NO\_MALLOC The allocation function IDAInit has not been 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 state vector y.

F2003 Name FIDASVtolerances

# IDAWFtolerances

Call flag = IDAWFtolerances(ida\_mem, efun);

Description 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 Eq. (2.6).

Arguments ida\_mem (void \*) pointer to the IDA memory block returned by IDACreate.

efun (IDAEwtFn) is the C function which defines the ewt vector (see §4.6.3).

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

IDA\_SUCCESS The call to IDAWFtolerances was successful.

 ${\tt IDA\_MEM\_NULL} \quad {\tt The\ IDA\ memory\ block\ was\ not\ initialized\ through\ a\ previous\ call\ to}$ 

IDACreate.

IDA\_NO\_MALLOC The allocation function IDAInit has not been called.

F2003 Name FIDAWFtolerances

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= $10^{-4}$  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 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 IDA package, and the discussion of it in the IDA Examples document [34]. In that problem, the three components vary between 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 a 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  $reltol=10^{-6}$ . 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.

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 IDA, with magnitude comparable to abstol or less, is equivalent to zero as far as the computation is concerned.
- (3) The user's residual routine **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) IDA 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.

#### 4.5.3 Linear solver interface functions

As previously explained, if the nonlinear solver requires the solution of linear systems of the form (2.4) (e.g., the default Newton iteration, then solution of these linear systems is handled with the IDALS linear solver interface. This interface supports all valid SUNLINSOL modules. Here, matrix-based SUNLINSOL modules utilize SUNMATRIX objects to store the Jacobian matrix  $J = \partial F/\partial y + \alpha \partial F/\partial \dot{y}$  and factorizations used throughout the solution process. Conversely, matrix-free SUNLINSOL modules instead use iterative methods to solve the linear systems of equations, and only require the *action* of the Jacobian on a vector, Jv.

With most iterative linear solvers, preconditioning can be done on the left only, on the right only, on both the left and the right, or not at all. The exceptions to this rule are SPFGMR that supports right preconditioning only and PCG that performs symmetric preconditioning. However, in IDA only left preconditioning is supported. For the specification of a preconditioner, see the iterative linear solver sections in  $\S4.5.8$  and  $\S4.6$ . A preconditioner matrix P must approximate the Jacobian J, at least crudely.

To specify a generic linear solver to IDA, after the call to IDACreate but before any calls to IDASolve, the user's program must create the appropriate SUNLINSOL object and call the function IDASetLinearSolver, as documented below. To create the SUNLinearSolver object, the user may call one of the SUNDIALS-packaged SUNLINSOL module constructor routines via a call of the form

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

The current list of such constructor routines includes SUNLinSol\_Dense, SUNLinSol\_Band, SUNLinSol\_LapackDense, SUNLinSol\_LapackBand, SUNLinSol\_KLU, SUNLinSol\_SUNLinSol\_SPECGMR, SUNLinSol\_SPECGMR, SUNLinSol\_SPECGMR, SUNLinSol\_SPECGMR, SUNLinSol\_SPECGMR, SUNLinSol\_SPECGMR, and SUNLinSol\_PCG.

Alternately, a user-supplied SUNLinearSolver module may be created and used instead. The use of each of the generic linear solvers involves certain constants, functions and possibly some macros, that are likely to be needed in the user code. These are available in the corresponding header file associated with the specific SUNMATRIX or SUNLINSOL module in question, as described in Chapters 8 and 9.

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

#### IDASetLinearSolver

Call flag = IDASetLinearSolver(ida\_mem, LS, J);

Description The function IDASetLinearSolver attaches a generic SUNLINSOL object LS and corresponding template Jacobian SUNMATRIX object J (if applicable) to IDA, initializing the IDALS linear solver interface.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

LS (SUNLinearSolver) SUNLINSOL object to use for solving linear systems of the form (2.4.

J (SUNMatrix) SUNMATRIX object for used as a template for the Jacobian (or NULL if not applicable).

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

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 current NVECTOR module.

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 type in Chapter 8 for further information).

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.

F2003 Name FIDASetLinearSolver

#### 4.5.4 Nonlinear solver interface function

By default IDA uses the SUNNONLINSOL implementation of Newton's method defined by the SUNNON-LINSOL\_NEWTON module (see §10.3). To specify a different nonlinear solver in IDA, the user's program must create a SUNNONLINSOL object by calling the appropriate constructor routine. The user must then attach the SUNNONLINSOL object to IDA by calling IDASetNonlinearSolver, as documented below.

When changing the nonlinear solver in IDA, IDASetNonlinearSolver must be called after IDAInit. If any calls to IDASolve have been made, then IDA 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 routine IDASetNonlinearSolver is the IDA memory pointer returned by IDACreate and the second argument is the SUNNONLINSOL object to use for solving the nonlinear system 2.3. A call to this function attaches the nonlinear solver to the main IDA integrator. We note that at present, the SUNNONLINSOL object must be of type SUNNONLINEARSOLVER\_ROOTFIND.

#### IDASetNonlinearSolver

Call flag = IDASetNonlinearSolver(ida\_mem, NLS);

Description The function IDASetNonLinearSolver attaches a SUNNONLINSOL object (NLS) to IDA.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

NLS (SUNNonlinearSolver) SUNNONLINSOL object to use for solving nonlinear systems.

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

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.

F2003 Name FIDASetNonlinearSolver

#### 4.5.5 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 Ref. [15].) It uses 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.

IDACalcIC

Call flag = IDACalcIC(ida\_mem, icopt, tout1);

Description The function IDACalcIC corrects the initial values y0 and yp0 at time t0.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

icopt (int) is one of the following two options for the initial condition calculation.

icopt=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.

icopt=IDA\_Y\_INIT directs IDACalcIC to compute all components of y, given  $\dot{y}$ . In this case, id is not required.

tout1 (realtype) 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 inte-

gration and rough scale in the independent variable t.

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

IDA\_SUCCESS IDASolve 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 man-

ner.

IDA\_LINIT\_FAIL The linear solver's initialization function failed.

IDA\_LSOLVE\_FAIL The linear solver's solve function failed in an unrecoverable man-

ner.

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.

IDALINESEARCH\_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.

IDACONV\_FAIL IDACalcIC failed to get convergence of the Newton iterations.

Notes All failure return values are negative and therefore a test flag < 0 will trap all IDACalcIC failures.

Note that 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 (see §4.5.10.3).

F2003 Name FIDACalcIC

# 4.5.6 Rootfinding initialization function

While integrating the IVP, IDA has the capability of finding the roots of a set of user-defined functions. To activate the rootfinding 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.

```
IDARootInit
```

Call flag = IDARootInit(ida\_mem, nrtfn, g);

Description The function IDARootInit specifies that the roots of a set of functions  $g_i(t, y, \dot{y})$  are to

be found while the IVP is being solved.

Arguments ida\_mem (void \*) pointer to the IDA memory block returned by IDACreate.

**nrtfn** (int) is the number of root functions  $g_i$ .

g (IDARootFn) is the C function which defines the nrtfn functions  $g_i(t,y,\dot{y})$ 

whose roots are sought. See §4.6.4 for details.

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

IDA\_SUCCESS The call to IDARootInit 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.

F2003 Name FIDARootInit

# 4.5.7 IDA solver function

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

# IDASolve

Call flag = IDASolve(ida\_mem, tout, &tret, yret, ypret, itask);

Description The function IDASolve integrates the DAE over an interval in t.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

tout (realtype) the next time at which a computed solution is desired.

tret (realtype) the time reached by the solver (output).

yret (N\_Vector) the computed solution vector y.

ypret (N\_Vector) the computed solution vector  $\dot{y}$ .

itask (int) a flag indicating the job of the solver for the next user 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 approximate values of y(tout) and  $\dot{y}(\texttt{tout})$ . The IDA\_ONE\_STEP option tells the solver to just take one internal step and return the solution at

the point reached by that step.

Return value IDASolve returns vectors yret and ypret and a corresponding independent variable value t = tret, such that (yret, ypret) are the computed values of  $(y(t), \dot{y}(t))$ .

In IDA\_NORMAL mode with no errors, tret will be equal to tout and yret = y(tout), ypret =  $\dot{y}(tout)$ .

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

IDA\_SUCCESS IDASolve succeeded.

IDA\_TSTOP\_RETURN IDASolve succeeded by reaching the stop point specified through

the optional input function IDASetStopTime. See §4.5.8.1 for more

information.

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. See §4.5.10.4 for more

information.

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: (a) The tolerances have not been set. (b) A component of the error weight vector became zero during internal time-stepping. (c) The linear solver initialization function (called by the user after calling IDACreate) failed to set the linear solver-specific lsolve field in ida\_mem. (d) 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.

 ${\tt IDA\_TOO\_MUCH\_ACC} \quad {\tt The \ solver \ could \ not \ satisfy \ the \ accuracy \ demanded \ by \ the \ user \ for \ accuracy \ demanded \ by \ the \ user \ for \ accuracy \ demanded \ by \ the \ user \ for \ accuracy \ demanded \ by \ the \ user \ for \ accuracy \ demanded \ by \ the \ user \ for \ accuracy \ demanded \ by \ the \ user \ for \ accuracy \ demanded \ by \ the \ user \ for \ accuracy \ demanded \ by \ the \ user \ for \ accuracy \ demanded \ by \ the \ user \ for \ accuracy \ demanded \ by \ the \ user \ for \ accuracy \ demanded \ by \ the \ user \ for \ accuracy \ demanded \ by \ the \ user \ for \ accuracy \ demanded \ by \ the \ user \ for \ accuracy \ demanded \ by \ the \ accuracy \ demanded \ by \ the \ user \ for \ accuracy \ demanded \ by \ the \ accuracy \ demanded \ by \ accuracy \ demanded \ de$ 

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 man-

ner.

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.

The vector yret can occupy the same space as the vector y0 of initial conditions that was passed to IDAInit, and the vector ypret can occupy the same space as yp0.

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  ${\tt flag} < 0$  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.

Notes

# 4.5.8 Optional input functions

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

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. Finally, a call to a IDASet\*\*\* function can be made from the user's calling program at any time and, if successful, takes effect immediately.

#### 4.5.8.1 Main solver optional input functions

The calls listed here can 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.

# IDASetErrFile

Call flag = IDASetErrFile(ida\_mem, errfp);

Description The function IDASetErrFile specifies the pointer to the file where all IDA messages

should be directed when the default IDA error handler function is used.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

errfp (FILE \*) pointer to output file.

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

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 IDA memory pointer is NULL). This use of IDASetErrFile is strongly discour-

aged.

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.

F2003 Name FIDASetErrFile

#### IDASetErrHandlerFn

Call flag = IDASetErrHandlerFn(ida\_mem, ehfun, eh\_data);

Description The function IDASetErrHandlerFn specifies the optional user-defined function to be

used in handling error messages.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

ehfun (IDAErrHandlerFn) is the user's C error handler function (see §4.6.2).

eh\_data (void \*) pointer to user data passed to ehfun every time it is called.

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

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 IDA solver memory is NULL will always be directed

to stderr.

F2003 Name FIDASetErrHandlerFn



Table 4.2: Optional inputs for IDA and IDALS

Optional input	Function name	Default				
IDA main solver						
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 $t_{\text{out}}$	${\tt IDASetMaxNumSteps}$	500				
Initial step size	IDASetInitStep	estimated				
Maximum absolute step size	${\tt IDASetMaxStep}$	$\infty$				
Value of $t_{stop}$	IDASetStopTime	$\infty$				
Maximum no. of error test failures	${\tt IDASetMaxErrTestFails}$	10				
Maximum no. of nonlinear iterations	${\tt IDASetMaxNonlinIters}$	4				
Maximum no. of convergence failures	${\tt IDASetMaxConvFails}$	10				
Coeff. in the nonlinear convergence test	IDASetNonlinConvCoef	0.33				
Suppress alg. vars. from error test	IDASetSuppressAlg	SUNFALSE				
Variable types (differential/algebraic)	IDASetId	NULL				
Inequality constraints on solution	IDASetConstraints	NULL				
Direction of zero-crossing	${\tt IDASetRootDirection}$	both				
Disable rootfinding warnings	${\tt IDASetNoInactiveRootWarn}$	none				
IDA initial conditions calculation						
Coeff. in the nonlinear convergence test	${\tt IDASetNonlinConvCoefIC}$	0.0033				
Maximum no. of steps	${\tt IDASetMaxNumStepsIC}$	5				
Maximum no. of Jacobian/precond. evals.	${\tt IDASetMaxNumJacsIC}$	4				
Maximum no. of Newton iterations	${\tt IDASetMaxNumItersIC}$	10				
Max. linesearch backtracks per Newton iter.	${\tt IDASetMaxBacksIC}$	100				
Turn off linesearch	IDASetLineSearchOffIC	SUNFALSE				
Lower bound on Newton step	${\tt IDASetStepToleranceIC}$	$uround^{2/3}$				
IDALS linear solver interface						
Jacobian function	IDASetJacFn	DQ				
Enable or disable linear solution scaling	${\tt IDASetLinearSolutionScaling}$	on				
Jacobian-times-vector function	IDASetJacTimes	NULL, DQ				
Preconditioner functions	IDASetPreconditioner	NULL, NULL				
Ratio between linear and nonlinear tolerances	${\tt IDASetEpsLin}$	0.05				
Increment factor used in DQ $Jv$ approx.	${\tt IDASetIncrementFactor}$	1.0				
Jacobian-times-vector DQ Res function	${\tt IDASetJacTimesResFn}$	NULL				
Newton linear solve tolerance conversion factor	IDASetLSNormFactor	vector length				

#### IDASetUserData

Call flag = IDASetUserData(ida\_mem, user\_data);

Description The function IDASetUserData specifies the user data block user\_data and attaches it

to the main IDA memory block.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

user\_data (void \*) pointer to the user data.

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

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.

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.

F2003 Name FIDASetUserData

# IDASetMaxOrd

Call flag = IDASetMaxOrd(ida\_mem, maxord);

Description The function IDASetMaxOrd specifies the maximum order of the linear multistep method.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

maxord (int) value of the maximum method order. This must be positive.

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

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 its previous value.

Notes The default value is 5. If the input value exceeds 5, the value 5 will be used. Since

maxord affects the memory requirements for the internal IDA memory block, its value

cannot be increased past its previous value.

 $F2003 \; \mathrm{Name} \; \; \mathtt{FIDASetMaxOrd}$ 

# IDASetMaxNumSteps

Call flag = IDASetMaxNumSteps(ida\_mem, mxsteps);

Description 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 (void \*) pointer to the IDA memory block.

mxsteps (long int) maximum allowed number of steps.

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

IDA\_SUCCESS The optional value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

Notes Passing mxsteps = 0 results in IDA using the default value (500).

Passing mxsteps < 0 disables the test (not recommended).

F2003 Name FIDASetMaxNumSteps



#### IDASetInitStep

Call flag = IDASetInitStep(ida\_mem, hin);

Description The function IDASetInitStep specifies the initial step size.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

hin (realtype) value of the initial step size to be attempted. Pass 0.0 to have IDA use the default value.

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

IDA\_SUCCESS The optional value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

Notes By default, IDA estimates the initial step as the solution of  $||h\dot{y}||_{WRMS} = 1/2$ , with an

added restriction that |h| < .001|tout - t0|.

F2003 Name FIDASetInitStep

#### IDASetMaxStep

Call flag = IDASetMaxStep(ida\_mem, hmax);

Description The function IDASetMaxStep specifies the maximum absolute value of the step size.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

hmax (realtype) maximum absolute value of the step size.

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

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  $\mathtt{hmax} = 0$  to obtain the default value  $\infty$ .

F2003 Name FIDASetMaxStep

# ${\tt IDASetStopTime}$

Call flag = IDASetStopTime(ida\_mem, tstop);

Description The function IDASetStopTime specifies the value of the independent variable t past

which the solution is not to proceed.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

tstop (realtype) value of the independent variable past which the solution should not proceed.

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

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).

F2003 Name FIDASetStopTime

#### IDASetMaxErrTestFails

Call flag = IDASetMaxErrTestFails(ida\_mem, maxnef);

Description The function IDASetMaxErrTestFails specifies the maximum number of error test

failures in attempting one step.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

maxnef (int) maximum number of error test failures allowed on one step (>0).

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

IDA\_SUCCESS The optional value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

Notes The default value is 10. F2003 Name FIDASetMaxErrTestFails

# IDASetMaxNonlinIters

Call flag = IDASetMaxNonlinIters(ida\_mem, maxcor);

Description The function IDASetMaxNonlinIters specifies the maximum number of nonlinear solver

iterations at one step.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

maxcor (int) maximum number of nonlinear solver iterations allowed on one step

(>0).

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

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 4.

F2003 Name FIDASetMaxNonlinIters

# IDASetMaxConvFails

Call flag = IDASetMaxConvFails(ida\_mem, maxncf);

Description The function IDASetMaxConvFails specifies the maximum number of nonlinear solver

convergence failures at one step.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

maxncf (int) maximum number of allowable nonlinear solver convergence failures on

one step (>0).

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

IDA\_SUCCESS The optional value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

Notes The default value is 10. F2003 Name FIDASetMaxConvFails

# IDASetNonlinConvCoef

Call flag = IDASetNonlinConvCoef(ida\_mem, nlscoef);

Description The function IDASetNonlinConvCoef specifies the safety factor in the nonlinear con-

vergence test; see Chapter 2, Eq. (2.7).

Arguments ida\_mem (void \*) pointer to the IDA memory block.

nlscoef (realtype) coefficient in nonlinear convergence test (> 0.0).

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

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  $\leq 0.0$ .

Notes The default value is 0.33.

F2003 Name FIDASetNonlinConvCoef

# ${\tt IDASetSuppressAlg}$

Call flag = IDASetSuppressAlg(ida\_mem, suppressalg);

Description The function IDASetSuppressAlg indicates whether or not to suppress algebraic vari-

ables in the local error test.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

suppressalg (booleantype) indicates whether to suppress (SUNTRUE) or not (SUNFALSE)

the algebraic variables in the local error test.

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

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 Ref. [11] for more on this issue.

F2003 Name FIDASetSuppressAlg

#### IDASetId

Call flag = IDASetId(ida\_mem, id);

Description The function IDASetId specifies algebraic/differential components in the y vector.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

 $\verb"id" (N\_Vector") state vector. A value of 1.0 indicates a differential variable, while$ 

0.0 indicates an algebraic variable.

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

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 lo-

cal error test (see IDASetSuppressAlg) or if IDACalcIC is to be called with icopt =

 $IDA_YA_YDP_INIT$  (see §4.5.5).

F2003 Name FIDASetId

#### IDASetConstraints

Call flag = IDASetConstraints(ida\_mem, constraints);

Description The function IDASetConstraints specifies a vector defining inequality constraints for

each component of the solution vector y.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

constraints (N\_Vector) vector of constraint flags. If constraints[i] is

0.0 then no constraint is imposed on  $y_i$ .

1.0 then  $y_i$  will be constrained to be  $y_i \ge 0.0$ .

-1.0 then  $y_i$  will be constrained to be  $y_i \leq 0.0$ .

2.0 then  $y_i$  will be constrained to be  $y_i > 0.0$ .

-2.0 then  $y_i$  will be constrained to be  $y_i < 0.0$ .

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

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.

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 will result in an illegal input return.

F2003 Name FIDASetConstraints

#### 4.5.8.2 Linear solver interface optional input functions

The mathematical explanation of the linear solver methods available to IDA 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 a dense or banded 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.

# IDASetJacFn

Call flag = IDASetJacFn(ida\_mem, jac);

Description The function IDASetJacFn specifies the Jacobian approximation function to be used for

a matrix-based solver within the IDALs interface.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

jac (IDALsJacFn) user-defined Jacobian approximation function.

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

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 dense and band matrices. If NULL is passed to jac, this default function is used. An error will occur if no jac is supplied when using other matrix types.

The function type IDALsJacFn is described in §4.6.5.

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.

#### F2003 Name FIDASetJacFn

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  $\alpha$  may not be current and a scaling factor is applied to the solution of the linear system to account for the lagged value of  $\alpha$ . See §9.4.1 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  $\alpha$  as part of the solve.

#### IDASetLinearSolutionScaling

Call flag = IDASetLinearSolutionScaling(ida\_mem, onoff);

Description The function IDASetLinearSolutionScaling enables or disables scaling the linear system solution to account for a change in  $\alpha$  in the linear system. For more details see

§9.4.1.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

onoff (booleantype) flag to enable (SUNTRUE) or disable (SUNFALSE) scaling

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

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.

#### F2003 Name FIDASetLinearSolutionScaling

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) 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 function must be of type IDALsJacTimesVecFn and can be specified through a call to IDASetJacTimes (see §4.6.6 for specification details). The evaluation and processing of any Jacobian-related data needed by the user's Jacobian-times-vector function may be done in the optional user-supplied function jtsetup (see §4.6.7 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-times-vector 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.

#### IDASetJacTimes

Call flag = IDASetJacTimes(ida\_mem, jsetup, jtimes);

Description The function IDASetJacTimes specifies the Jacobian-vector setup and product func-

tions.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

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

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

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

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 SUNLINSOL 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.

The function type IDALsJacTimesSetupFn is described in §4.6.7.

The function type IDALsJacTimesVecFn is described in §4.6.6.

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.

#### F2003 Name FIDASetJacTimes

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.

#### IDASetIncrementFactor

Call flag = IDASetIncrementFactor(ida\_mem, dqincfac);

Description 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

$$Jv = \frac{1}{\sigma} \left[ F(t, \tilde{y}, \tilde{y}') - F(t, y, y') \right],$$

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

Arguments ida\_mem (void \*) pointer to the IDA memory block.

dgincfac (realtype) user-specified increment factor (positive).

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

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  $\leq 0$ .

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.

> 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.

#### F2003 Name FIDASetIncrementFactor

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 [26] 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.

#### IDASetJacTimesResFn

Call flag = IDASetJacTimesResFn(ida\_mem, jtimesResFn);

Description The function IDASetJacTimesResFn specifies an alternative DAE residual function for

use in the internal Jacobian-vector product difference quotient approximation.

Arguments (void \*) pointer to the IDA memory block. ida\_mem

> jtimesResFn (IDAResFn) is the C function which computes the alternative DAE residual function to use in Jacobian-vector product difference quotient ap-

proximations. This function has the form res(t, yy, yp, resval, user\_data). For full details see §4.6.1.

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

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.

# F2003 Name FIDASetJacTimesResFn

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 IDA 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 §4.6. The user data pointer received through IDASetUserData (or a pointer to NULL if user data was not specified) is passed to the psetup and psolve functions. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied preconditioner functions without using global data in the program.

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

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

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

#### IDASetPreconditioner

Call flag = IDASetPreconditioner(ida\_mem, psetup, psolve);

Description The function IDASetPreconditioner specifies the preconditioner setup and solve func-

tions.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

psetup (IDALsPrecSetupFn) user-defined function to set up the preconditioner. Pass

NULL if no setup is necessary.

psolve (IDALsPrecSolveFn) user-defined preconditioner solve function.

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

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 SUNLINSOL 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.

The function type IDALsPrecSolveFn is described in §4.6.8.

The function type IDALsPrecSetupFn is described in §4.6.9.

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.

F2003 Name FIDASetPreconditioner

#### IDASetEpsLin

Call flag = IDASetEpsLin(ida\_mem, eplifac);

Description 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 (void \*) pointer to the IDA memory block.

eplifac (realtype) linear convergence safety factor ( $\geq 0.0$ ).

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

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.

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.

F2003 Name FIDASetEpsLin

#### IDASetLSNormFactor

Call flag = IDASetLSNormFactor(ida\_mem, nrmfac);

Description 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 idamem (void \*) pointer to the IDA memory block.

nrmfac (realtype) 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.</pre>

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

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 (IDA v5.0.0) the value of

nrmfac was computed using the vector dot product i.e., the nrmfac < 0 case.

F2003 Name FIDASetLSNormFactor

#### 4.5.8.3 Initial condition calculation optional input functions

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

#### IDASetNonlinConvCoefIC

Call flag = IDASetNonlinConvCoefIC(ida\_mem, epiccon);

Description The function IDASetNonlinConvCoefIC specifies the positive constant in the Newton

iteration convergence test within the initial condition calculation.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

epiccon (realtype) coefficient in the Newton convergence test (>0).

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

IDA\_SUCCESS The optional value has been successfully set.

 ${\tt IDA\_MEM\_NULL} \quad {\tt The \ ida\_mem \ pointer \ is \ NULL}.$ 

IDA\_ILL\_INPUT The epiccon factor is  $\leq 0.0$ .

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.

F2003 Name FIDASetNonlinConvCoefIC

#### IDASetMaxNumStepsIC

Call flag = IDASetMaxNumStepsIC(ida\_mem, maxnh);

Description 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,

 $J = \partial F/\partial y + (1/h)\partial F/\partial \dot{y}.$ 

Arguments ida\_mem (void \*) pointer to the IDA memory block.

maxnh (int) maximum allowed number of values for h.

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

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.

F2003 Name FIDASetMaxNumStepsIC

# IDASetMaxNumJacsIC

Call flag = IDASetMaxNumJacsIC(ida\_mem, maxnj);

Description 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 (void \*) pointer to the IDA memory block.

maxnj (int) maximum allowed number of Jacobian or preconditioner evaluations.

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

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.

F2003 Name FIDASetMaxNumJacsIC

# IDASetMaxNumItersIC

Call flag = IDASetMaxNumItersIC(ida\_mem, maxnit);

Description The function IDASetMaxNumItersIC specifies the maximum number of Newton itera-

tions allowed in any one attempt to solve the initial conditions calculation problem.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

maxnit (int) maximum number of Newton iterations.

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

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.

F2003 Name FIDASetMaxNumItersIC

#### IDASetMaxBacksIC

Call flag = IDASetMaxBacksIC(ida\_mem, maxbacks);

Description The function IDASetMaxBacksIC specifies the maximum number of linesearch back-

tracks allowed in any Newton iteration, when solving the initial conditions calculation

problem.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

maxbacks (int) maximum number of linesearch backtracks per Newton step.

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

IDA\_SUCCESS The optional value has been successfully set.

 $\label{local_norm} \mbox{IDA\_MEM\_NULL} \quad \mbox{The ida\_mem pointer is NULL}.$ 

IDA\_ILL\_INPUT maxbacks is non-positive.

Notes The default value is 100.

F2003 Name FIDASetMaxBacksIC

#### IDASetLineSearchOffIC

Call flag = IDASetLineSearchOffIC(ida\_mem, lsoff);

Description The function IDASetLineSearchOffIC specifies whether to turn on or off the linesearch

algorithm.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

 ${\tt lsoff} \quad ({\tt booleantype}) \ {\rm a} \ {\rm flag} \ {\rm to} \ {\rm turn} \ {\rm off} \ ({\tt SUNTRUE}) \ {\rm or} \ {\rm keep} \ ({\tt SUNFALSE}) \ {\rm the} \ {\rm linesearch}$ 

algorithm.

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

IDA\_SUCCESS The optional value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

Notes The default value is SUNFALSE.

F2003 Name FIDASetLineSearchOffIC

# IDASetStepToleranceIC

Call flag = IDASetStepToleranceIC(ida\_mem, steptol);

Description The function IDASetStepToleranceIC specifies a positive lower bound on the Newton

step.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

steptol (int) Minimum allowed WRMS-norm of the Newton step (> 0.0).

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

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  $\leq 0.0$ .

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

F2003 Name FIDASetStepToleranceIC

# 4.5.8.4 Rootfinding optional input functions

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

#### IDASetRootDirection

Call flag = IDASetRootDirection(ida\_mem, rootdir);

Description The function IDASetRootDirection specifies the direction of zero-crossings to be lo-

cated and returned to the user.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

rootdir (int \*) 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 indicates that the solver should report only zero-crossings where

 $g_i$  is increasing or decreasing, respectively.

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

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.

F2003 Name FIDASetRootDirection

#### IDASetNoInactiveRootWarn

Call flag = IDASetNoInactiveRootWarn(ida\_mem);

Description The function IDASetNoInactiveRootWarn disables issuing a warning if some root func-

tion appears to be identically zero at the beginning of the integration.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

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

IDA\_SUCCESS The optional value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

Notes IDA 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), IDA will issue a warning which can be disabled with this optional input function.

F2003 Name FIDASetNoInactiveRootWarn

# 4.5.9 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 IDA.

The call to the IDAGetDky function has the following form:

# ${\tt IDAGetDky}$

Call flag = IDAGetDky(ida\_mem, t, k, dky);

Description 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 IDA. 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  $t_n - h_u \le t \le t_n$ , where  $t_n$  denotes the current

internal time reached, and  $h_u$  is the last internal step size used successfully.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

t (realtype) time at which to interpolate.

```
k (int) integer specifying the order of the derivative of y wanted.

dky (N_Vector) vector containing the interpolated k^{th} derivative of y(t).
```

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

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  $\{0, 1, \ldots, klast\}$ .

IDA\_BAD\_DKY dky is NULL.

Notes It is only legal to call the function IDAGetDky after a successful return from IDASolve.

 $Functions \ \mathtt{IDAGetCurrentTime}, \ \mathtt{IDAGetLastStep} \ \mathrm{and} \ \mathtt{IDAGetLastOrder} \ (\mathrm{see} \ \S 4.5.10.2)$ 

can be used to access  $t_n$ ,  $h_u$  and klast.

F2003 Name FIDAGetDky

# 4.5.10 Optional output functions

IDA provides an extensive list of functions that can be used to obtain solver performance information. Table 4.3 lists all optional output functions in IDA, 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 IDA 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.

#### 4.5.10.1 SUNDIALS version information

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

# SUNDIALSGetVersion

Call flag = SUNDIALSGetVersion(version, len);

Description The function SUNDIALSGetVersion fills a character array with SUNDIALS version infor-

mation.

Arguments version (char \*) character array to hold the SUNDIALS version information.

len (int) allocated length of the version character array.

Return value If successful, SUNDIALSGetVersion returns 0 and version contains the SUNDIALS ver-

sion information. Otherwise, it returns -1 and version is not set (the input character

array is too short).

Notes A string of 25 characters should be sufficient to hold the version information. Any

trailing characters in the version array are removed.

Table 4.3: Optional outputs from IDA and IDALS

Optional output	Function name					
IDA main solver						
Size of IDA real and integer workspace	IDAGetWorkSpace					
Cumulative number of internal steps	IDAGetNumSteps					
No. of calls to residual function	IDAGetNumResEvals					
No. of calls to linear solver setup function	${\tt IDAGetNumLinSolvSetups}$					
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					
Order reductions due to stability limit detection	IDAGetNumStabLimOrderReds					
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					
IDA initial conditions calcu						
Number of backtrack operations	IDAGetNumBacktrackops					
Corrected initial conditions	IDAGetConsistentIC					
IDALS linear solver interface						
Size of real and integer workspace	${\tt IDAGetLinWorkSpace}$					
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					

# SUNDIALSGetVersionNumber

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

Description The function SUNDIALSGetVersionNumber set integers for the SUNDIALS major, minor,

and patch release numbers and fills a character array with the release label if applicable.

Arguments major (int) SUNDIALS release major version number.

minor (int) SUNDIALS release minor version number.

patch (int) SUNDIALS release patch version number.

label (char \*) character array to hold the SUNDIALS release label.

len (int) allocated length of the label character array.

 $Return\ value\ If\ successful,\ {\tt SUNDIALSGetVersionNumber}\ returns\ 0\ and\ the\ {\tt major},\ {\tt minor},\ {\tt patch},\ and$ 

label values are set. Otherwise, it returns -1 and the values are not set (the input

character array is too short).

Notes A string of 10 characters should be sufficient to hold the label information. If a label

is not used in the release version, no information is copied to label. Any trailing

characters in the label array are removed.

# 4.5.10.2 Main solver optional output functions

IDA 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 IDA memory block (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 SUNNONLINSOL nonlinear solver being used. As a convenience, additional extraction functions provide the optional outputs in groups. These optional output functions are described next.

# IDAGetWorkSpace

Call flag = IDAGetWorkSpace(ida\_mem, &lenrw, &leniw);

Description The function IDAGetWorkSpace returns the IDA real and integer workspace sizes.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

lenrw (long int) number of real values in the IDA workspace.

leniw (long int) number of integer values in the IDA workspace.

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

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 nrtfn of root functions (see §4.5.6), 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  $m = \max(\max, 3)$ , 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 =  $38 + (m+6) * N_i + \text{nrtfn}$ ;

- with IDASVtolerances: leniw = leniw  $+N_i$ ;
- with constraint checking: lenrw = lenrw + $N_i$ ;
- with id specified: lenrw = lenrw  $+N_i$ ;

where  $N_i$  is the number of integer words in one N\_Vector (= 1 for NVECTOR\_SERIAL and 2\*npes for NVECTOR\_PARALLEL 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 + 11N, leniw = 49.

F2003 Name FIDAGetWorkSpace

# IDAGetNumSteps

Call flag = IDAGetNumSteps(ida\_mem, &nsteps);

 $\label{prop:local_decomposition} \textbf{DagetNumSteps} \ \ \text{returns} \ \ \text{the cumulative number of internal steps} \ \ \text{taken}$ 

by the solver (total so far).

Arguments ida\_mem (void \*) pointer to the IDA memory block.

nsteps (long int) number of steps taken by IDA.

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

F2003 Name FIDAGetNumSteps

# IDAGetNumResEvals

Call flag = IDAGetNumResEvals(ida\_mem, &nrevals);

Description The function IDAGetNumResEvals returns the number of calls to the user's residual

evaluation function.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

nrevals (long int) number of calls to the user's res function.

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

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.

 $F2003 \; \mathrm{Name} \; \; \mathtt{FIDAGetNumResEvals}$ 

# IDAGetNumLinSolvSetups

Call flag = IDAGetNumLinSolvSetups(ida\_mem, &nlinsetups);

Description The function IDAGetNumLinSolvSetups returns the cumulative number of calls made

to the linear solver's setup function (total so far).

Arguments ida\_mem (void \*) pointer to the IDA memory block.

nlinsetups (long int) number of calls made to the linear solver setup function.

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

F2003 Name FIDAGetNumLinSolvSetups

# IDAGetNumErrTestFails

Call flag = IDAGetNumErrTestFails(ida\_mem, &netfails);

Description The function IDAGetNumErrTestFails returns the cumulative number of local error

test failures that have occurred (total so far).

Arguments ida\_mem (void \*) pointer to the IDA memory block.

netfails (long int) number of error test failures.

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

F2003 Name FIDAGetNumErrTestFails

# IDAGetLastOrder

Call flag = IDAGetLastOrder(ida\_mem, &klast);

Description The function IDAGetLastOrder returns the integration method order used during the

last internal step.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

klast (int) method order used on the last internal step.

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

F2003 Name FIDAGetLastOrder

# IDAGetCurrentOrder

Call flag = IDAGetCurrentOrder(ida\_mem, &kcur);

Description The function IDAGetCurrentOrder returns the integration method order to be used on

the next internal step.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

kcur (int) method order to be used on the next internal step.

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

F2003 Name FIDAGetCurrentOrder

# IDAGetLastStep

Call flag = IDAGetLastStep(ida\_mem, &hlast);

 $\label{prop:local_def} \textbf{Description} \quad \text{The function $\mathtt{IDAGetLastStep}$ 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 (void \*) pointer to the IDA memory block.

hlast (realtype) step size taken on the last internal step by IDA, or last artificial

step size used in IDACalcIC, whichever was called last.

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

F2003 Name FIDAGetLastStep

# IDAGetCurrentStep

Call flag = IDAGetCurrentStep(ida\_mem, &hcur);

Description The function IDAGetCurrentStep returns the integration step size to be attempted on

the next internal step.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

hcur (realtype) step size to be attempted on the next internal step.

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

F2003 Name FIDAGetCurrentStep

# IDAGetActualInitStep

Call flag = IDAGetActualInitStep(ida\_mem, &hinused);

Description The function IDAGetActualInitStep returns the value of the integration step size used

on the first step.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

hinused (realtype) actual value of initial step size.

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

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 IDA 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.

F2003 Name FIDAGetActualInitStep

#### IDAGetCurrentTime

Call flag = IDAGetCurrentTime(ida\_mem, &tcur);

Description The function IDAGetCurrentTime returns the current internal time reached by the

solver.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

tcur (realtype) current internal time reached.

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

F2003 Name FIDAGetCurrentTime

# IDAGetTolScaleFactor

Call flag = IDAGetTolScaleFactor(ida\_mem, &tolsfac);

Description 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 (void \*) pointer to the IDA memory block.

tolsfac (realtype) suggested scaling factor for user tolerances.

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

F2003 Name FIDAGetTolScaleFactor

# IDAGetErrWeights

Call flag = IDAGetErrWeights(ida\_mem, eweight);

Description The function IDAGetErrWeights returns the solution error weights at the current time.

These are the  $W_i$  given by Eq. (2.6) (or by the user's IDAEwtFn).

Arguments ida\_mem (void \*) pointer to the IDA memory block.

eweight (N\_Vector) solution error weights at the current time.

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

Notes The user must allocate space for eweight.

F2003 Name FIDAGetErrWeights

# IDAGetEstLocalErrors

Call flag = IDAGetEstLocalErrors(ida\_mem, ele);

Description The function IDAGetEstLocalErrors returns the estimated local errors.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

ele (N\_Vector) estimated local errors at the current time.

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

Notes The user must allocate space for ele.

The values returned in ele are only valid if IDASolve returned a non-negative value.

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].

 $F2003 \ \mathrm{Name}$  FIDAGetEstLocalErrors

# IDAGetIntegratorStats

Call flag = IDAGetIntegratorStats(ida\_mem, &nsteps, &nrevals, &nlinsetups, &netfails, &klast, &kcur, &hinused,

&hlast, &hcur, &tcur);

Description The function IDAGetIntegratorStats returns the IDA integrator statistics as a group.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

nsteps (long int) cumulative number of steps taken by IDA.

nrevals (long int) cumulative number of calls to the user's res function.

nlinsetups (long int) cumulative number of calls made to the linear solver setup function.





netfails (long int) cumulative number of error test failures.klast (int) method order used on the last internal step.kcur (int) method order to be used on the next internal step.

hinused (realtype) actual value of initial step size.

hlast (realtype) step size taken on the last internal step.

hcur (realtype) step size to be attempted on the next internal step.

tcur (realtype) current internal time reached.

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

IDA\_SUCCESS the optional output values have been successfully set.

IDA\_MEM\_NULL the ida\_mem pointer is NULL.

F2003 Name FIDAGetIntegratorStats

# IDAGetNumNonlinSolvIters

Call flag = IDAGetNumNonlinSolvIters(ida\_mem, &nniters);

Description The function IDAGetNumNonlinSolvIters returns the cumulative number of nonlinear

iterations performed.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

nniters (long int) number of nonlinear iterations performed.

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

IDA\_MEM\_FAIL The SUNNONLINSOL module is NULL.

F2003 Name FIDAGetNumNonlinSolvIters

# IDAGetNumNonlinSolvConvFails

Call flag = IDAGetNumNonlinSolvConvFails(ida\_mem, &nncfails);

Description The function IDAGetNumNonlinSolvConvFails returns the cumulative number of non-

linear convergence failures that have occurred.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

nncfails (long int) number of nonlinear convergence failures.

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

F2003 Name FIDAGetNumNonlinSolvConvFails

# IDAGetNonlinSolvStats

Call flag = IDAGetNonlinSolvStats(ida.mem, &nniters, &nncfails);

 $\label{thm:decomposition} \textbf{Description} \quad \textbf{The function IDAGetNonlinSolvStats} \ \ \textbf{returns} \ \ \textbf{the IDA} \ \ \textbf{nonlinear solver} \ \ \textbf{statistics} \ \ \textbf{as} \ \ \textbf{a}$ 

group.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

nniters (long int) cumulative number of nonlinear iterations performed. nncfails (long int) cumulative number of nonlinear convergence failures.

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

IDA\_MEM\_FAIL The SUNNONLINSOL module is NULL.

F2003 Name FIDAGetNonlinSolvStats

# ${\tt IDAGetReturnFlagName}$

Description The function IDAGetReturnFlagName returns the name of the IDA constant correspond-

ing to flag.

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

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

F2003 Name FIDAGetReturnFlagName

# 4.5.10.3 Initial condition calculation optional output functions

# IDAGetNumBcktrackOps

Call flag = IDAGetNumBacktrackOps(ida\_mem, &nbacktr);

Description The function IDAGetNumBacktrackOps returns the number of backtrack operations done

in the linesearch algorithm in IDACalcIC.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

nbacktr (long int) the cumulative number of backtrack operations.

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

F2003 Name FIDAGetNumBcktrackOps

# IDAGetConsistentIC

Call flag = IDAGetConsistentIC(ida\_mem, yy0\_mod, yp0\_mod);

Description The function IDAGetConsistentIC returns the corrected initial conditions calculated

by IDACalcIC.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

 $\verb|yy0_mod| (\verb|N_Vector|) consistent solution vector.$ 

yp0\_mod (N\_Vector) consistent derivative vector.

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

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.

The user must allocate space for yyo\_mod and ypo\_mod (if not NULL).

F2003 Name FIDAGetConsistentIC

### 4.5.10.4 Rootfinding optional output functions

There are two optional output functions associated with rootfinding.



# IDAGetRootInfo

Call flag = IDAGetRootInfo(ida\_mem, rootsfound);

Description The function IDAGetRootInfo returns an array showing which functions were found to

have a root.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

rootsfound (int \*) array of length nrtfn with the indices of the user functions  $g_i$  found to have a root. For i = 0, ..., nrtfn -1, rootsfound[i]  $\neq 0$  if  $g_i$  has a

root, and = 0 if not.

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

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  $g_i$ .

The user must allocate memory for the vector rootsfound.

F2003 Name FIDAGetRootInfo

# IDAGetNumGEvals

Call flag = IDAGetNumGEvals(ida\_mem, &ngevals);

Description The function IDAGetNumgEvals returns the cumulative number of calls to the user root

function q.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

ngevals (long int) number of calls to the user's function g so far.

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

F2003 Name FIDAGetNumGEvals

# 4.5.10.5 IDALS linear solver interface optional output functions

The following optional outputs are available from the IDALS modules: workspace requirements, number of calls to the Jacobian routine, number of calls to the residual routine for finite-difference Jacobian or Jacobian-vector product approximation, number of linear iterations, number of linear convergence failures, number of calls to the preconditioner setup and solve routines, number of calls to the Jacobian-vector setup and product routines, and last return value from an IDALS function. Note that, where the name of an output would otherwise conflict with the name of an optional output from the main solver, a suffix LS (for Linear Solver) has been added (e.g., lenrwLS).

#### IDAGetLinWorkSpace

Call flag = IDAGetLinWorkSpace(ida\_mem, &lenrwLS, &leniwLS);

Description The function IDAGetLinWorkSpace returns the sizes of the real and integer workspaces

used by the IDALS linear solver interface.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

lenrwLS (long int) the number of real values in the IDALS workspace.

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

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



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

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.

F2003 Name FIDAGetLinWorkSpace

# IDAGetNumJacEvals

Call flag = IDAGetNumJacEvals(ida\_mem, &njevals);

Description The function IDAGetNumJacEvals returns the cumulative number of calls to the IDALS

Jacobian approximation function.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

njevals (long int) the cumulative number of calls to the Jacobian function (total so

far).

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

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 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.

 $F2003 \; \mathrm{Name} \; \; \mathtt{FIDAGetNumJacEvals}$ 

### IDAGetNumLinResEvals

Call flag = IDAGetNumLinResEvals(ida\_mem, &nrevalsLS);

Description 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 (void \*) pointer to the IDA memory block.

nrevalsLS (long int) the cumulative number of calls to the user residual function.

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

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.

The previous routines <code>IDADlsGetNumRhsEvals</code> and <code>IDASpilsGetNumRhsEvals</code> are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to

the new routine name soon.

 $F2003 \ \mathrm{Name} \ \mathrm{FIDAGetNumLinResEvals}$ 

# IDAGetNumLinIters

Call flag = IDAGetNumLinIters(ida\_mem, &nliters);

Description The function IDAGetNumLinIters returns the cumulative number of linear iterations.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

nliters (long int) the current number of linear iterations.

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

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 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.

F2003 Name FIDAGetNumLinIters

#### IDAGetNumLinConvFails

Call flag = IDAGetNumLinConvFails(ida\_mem, &nlcfails);

Description The function IDAGetNumLinConvFails returns the cumulative number of linear conver-

gence failures.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

nlcfails (long int) the current number of linear convergence failures.

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

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 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.

 $F2003 \; \mathrm{Name} \; \; \mathtt{FIDAGetNumLinConvFails}$ 

# IDAGetNumPrecEvals

Call flag = IDAGetNumPrecEvals(ida\_mem, &npevals);

Description The function IDAGetNumPrecEvals returns the cumulative number of preconditioner

evaluations, i.e., the number of calls made to psetup.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

npevals (long int) the cumulative number of calls to psetup.

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

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 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.

 $F2003 \ \mathrm{Name} \ \mathtt{FIDAGetNumPrecEvals}$ 

# IDAGetNumPrecSolves

Call flag = IDAGetNumPrecSolves(ida\_mem, &npsolves);

Description The function IDAGetNumPrecSolves returns the cumulative number of calls made to

the preconditioner solve function, psolve.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

npsolves (long int) the cumulative number of calls to psolve.

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

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.

The previous routine IDASpilsGetNumPrecSolves is now a wrapper for this routine, Notes

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.

F2003 Name FIDAGetNumPrecSolves

# IDAGetNumJTSetupEvals

Call flag = IDAGetNumJTSetupEvals(ida\_mem, &njtsetup);

Description The function IDAGetNumJTSetupEvals returns the cumulative number of calls made to

the Jacobian-vector setup function jtsetup.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

njtsetup (long int) the current number of calls to jtsetup.

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

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 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.

F2003 Name FIDAGetNumJTSetupEvals

#### IDAGetNumJtimesEvals

Call flag = IDAGetNumJtimesEvals(ida\_mem, &njvevals);

Description The function IDAGetNumJtimesEvals returns the cumulative number of calls made to

the Jacobian-vector function, jtimes.

ida\_mem (void \*) pointer to the IDA memory block. Arguments

njvevals (long int) the cumulative number of calls to jtimes.

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

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 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.

F2003 Name FIDAGetNumJtimesEvals

# IDAGetLastLinFlag

Call flag = IDAGetLastLinFlag(ida\_mem, &lsflag);

Description The function IDAGetLastLinFlag returns the last return value from an IDALS routine.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

lsflag (long int) the value of the last return flag from an IDALS function.

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

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 IDALSETUP\_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 SUNLINSOL module, 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 IDALSOLVE\_FAIL), lsflag contains the error return flag from the SUNLINSOL object, which will be one of:

 ${\tt SUNLS\_MEM\_NULL}, \ {\tt indicating} \ \ {\tt that} \ \ {\tt the} \ \ {\tt SUNLINSOL} \ \ {\tt memory} \ \ {\tt is} \ \ {\tt NULL};$ 

SUNLS\_ATIMES\_FAIL\_UNREC, indicating an unrecoverable failure in the J\*v function; SUNLS\_PSOLVE\_FAIL\_UNREC, indicating that the preconditioner solve function <code>psolve</code> failed unrecoverably; SUNLS\_GS\_FAIL, indicating a failure in the Gram-Schmidt procedure (generated only in <code>SPGMR</code> or <code>SPFGMR</code>); SUNLS\_QRSOL\_FAIL, indicating that the matrix R was found to be singular during the QR solve phase (<code>SPGMR</code> and <code>SPFGMR</code> only); or <code>SUNLS\_PACKAGE\_FAIL\_UNREC</code>, indicating an unrecoverable failure in an external iterative linear solver package.

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.

F2003 Name FIDAGetLastLinFlag

#### IDAGetLinReturnFlagName

Description The function IDAGetLinReturnFlagName returns the name of the IDALS constant cor-

responding to lsflag.

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

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

If  $1 \leq lsflag \leq N$  (LU factorization failed), this function returns "NONE".

Notes

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.

F2003 Name FIDAGetLinReturnFlagName

# 4.5.11 IDA reinitialization function

The function IDAReInit reinitializes the main IDA 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 NVECTOR 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 §4.5.3.

If there are changes to any optional inputs, make the appropriate IDASet\*\*\* calls, as described in §4.5.8. 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 tout. 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.

```
IDAReInit
Call
              flag = IDAReInit(ida_mem, t0, y0, yp0);
Description
             The function IDAReInit provides required problem specifications and reinitializes IDA.
Arguments
              ida_mem (void *) pointer to the IDA memory block.
                       (realtype) is the initial value of t.
              t0
                       (N_{\text{-}}Vector) is the initial value of y.
              y0
                       (N_Vector) is the initial value of \dot{y}.
              yp0
Return value The return value flag (of type int) will be one of the following:
              IDA_SUCCESS
                              The call to IDAReInit was successful.
              IDA_MEM_NULL
                             The IDA memory block was not initialized through a previous call to
                              IDACreate.
              IDA_NO_MALLOC Memory space for the IDA memory block was not allocated through a
                              previous call to IDAInit.
```

IDA\_ILL\_INPUT An input argument to IDAReInit has an illegal value.

If an error occurred, IDAReInit also sends an error message to the error handler func-

F2003 Name FIDAReInit

tion.

Notes

# 4.6 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.

# 4.6.1 Residual function

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

# IDAResFn

Definition typedef int (\*IDAResFn)(realtype tt, N\_Vector yy, N\_Vector yp, N\_Vector rr, void \*user\_data);

Purpose 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 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, IDA 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.)

Allocation of memory for yp is handled within IDA.

# 4.6.2 Error message handler function

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

# IDAErrHandlerFn

Definition typedef void (\*IDAErrHandlerFn)(int error\_code, const char \*module, const char \*function, char \*msg, void \*eh\_data);

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

Arguments error\_code is the error code.

module is the name of the IDA module reporting the error.

function is the name of the function in which the error occurred.

msg is the error message.

eh\_data is a pointer to user data, the same as the eh\_data parameter passed to

 ${\tt IDASetErrHandlerFn}.$ 

Return value A IDAErrHandlerFn 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.

# 4.6.3 Error weight function

As an alternative to providing the relative and absolute tolerances, the user may provide a function of type IDAEwtFn to compute a vector ewt containing the multiplicative weights  $W_i$  used in the WRMS norm  $||v||_{\text{WRMS}} = \sqrt{(1/N)\sum_1^N (W_i \cdot v_i)^2}$ . These weights will used in place of those defined by Eq. (2.6). The function type IDAEwtFn is defined as follows:

#### IDAEwtFn

Definition typedef int (\*IDAEwtFn)(N\_Vector y, N\_Vector ewt, void \*user\_data);

Purpose 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 An IDAEwtFn function type must return 0 if it successfully set the error weights and -1 otherwise.

Notes Allocation of memory for ewt is handled within IDA.

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

# 4.6.4 Rootfinding function

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

# IDARootFn

Notes

Definition typedef int (\*IDARootFn)(realtype t, N\_Vector y, N\_Vector yp, realtype \*gout, void \*user\_data);

Purpose 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 An IDARootFn should return 0 if successful or a non-zero value if an error occurred (in which case the integration is halted and IDASolve returns IDA\_RTFUNC\_FAIL).

Allocation of memory for gout is handled within IDA.

# 4.6.5 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:



# IDALsJacFn

Purpose This function computes the Jacobian matrix J of the DAE system (or an approximation to it), defined by Eq. (2.5).

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 ( $\alpha$  in Eq. (2.5)).

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),  $J = \partial F/\partial y + cj \partial F/\partial \dot{y}$ .

user\_data is a pointer to user data, the same as the user\_data parameter passed to IDASetUserData.

tmp1

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 error return, the integrator will attempt to recover by reducing the stepsize, and hence changing  $\alpha$  in (2.5).

Notes

Information regarding the structure of the specific SUNMATRIX structure (e.g., number of rows, upper/lower bandwidth, sparsity type) may be obtained through using the implementation-specific SUNMATRIX interface functions (see Chapter 8 for details).

With direct linear solvers (i.e., linear solvers with type SUNLINEARSOLVER\_DIRECT), the Jacobian matrix J(t,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 Netwon method), each call to the user's IDALsJacFn function is preceded by a call to the IDAResFn user function with the same (tt, yy, yp) 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 §10.1.4 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 §4.5.10.2. 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...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...N - 1), and the elements of the j-th column can then be accessed using ordinary array indexing. Consequently,  $J_{m,n}$  can be loaded using the statements col\_n = SM\_COLUMN\_D(J, n-1); col\_n[m-1] =  $J_{m,n}$ . For large problems, it is more efficient to use SM\_COLUMN\_D than to use SM\_ELEMENT\_D. Note that both of these macros number rows and columns starting from 0. The SUNMATRIX\_DENSE type and accessor macros are documented in §8.3.

#### banded:

A user-supplied banded Jacobian function must load the Neq × 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 < m-n < 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 §8.4.

#### 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 §8.5.

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.

# 4.6.6 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.

# IDALsJacTimesVecFn

Definition 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);

Purpose This function computes the product Jv of the DAE system Jacobian J (or an approximation to it) and a given vector  $\mathbf{v}$ , where J is defined by Eq. (2.5).

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  $size (\alpha \text{ in } E\alpha, (2.5))$ 

size ( $\alpha$  in Eq. (2.5)).

user\_data is a pointer to user data, the same as the user\_data parameter passed to IDASetUserData.

tmp1

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 J \* v that uses 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 §4.5.10.2. The unit roundoff can be accessed as UNIT\_ROUNDOFF defined in sundials\_types.h.

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.

# 4.6.7 Jacobian-vector product setup (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 IDALsJacTimesSetupFn, defined as follows:

# ${\tt IDALsJacTimesSetupFn}$

Purpose This function preprocesses and/or evaluates Jacobian data needed by the Jacobian-

times-vector routine.

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 ( $\alpha$  in Eq. (2.5)).

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 setup function is preceded by a call to the IDAResFn user function with the same (t,y, yp) 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 §4.5.10.2. The unit roundoff can be accessed as UNIT\_ROUNDOFF defined in sundials\_types.h.

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.

# 4.6.8 Preconditioner solve (iterative linear solvers)

If a user-supplied preconditioner is to be used with a SUNLINSOL solver module, then the user must provide a function to solve the linear system Pz = r where P is 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:

```
IDALsPrecSolveFn
Definition
              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);
Purpose
              This function solves the preconditioning system Pz = r.
Arguments
              t.t.
                          is the current value of the independent variable.
                          is the current value of the dependent variable vector, y(t).
              уу
                          is the current value of \dot{y}(t).
              ур
              rr
                          is the current value of the residual vector F(t, y, \dot{y}).
                          is the right-hand side vector r of the linear system to be solved.
              rvec
                          is the computed output vector.
              zvec
                          is the scalar in the system Jacobian, proportional to the inverse of the step
              сj
                          size (\alpha in Eq. (2.5)).
              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}
```

the function IDASetUserData.

delta. To obtain the N\_Vector ewt, call IDAGetErrWeights (see §4.5.10.2).

user\_data is a pointer to user data, the same as the user\_data parameter passed to

Return value The value to be returned by the preconditioner solve function is a flag indicating whether

it was successful. This value should be 0 if successful, positive for a recoverable error (in which case the step will be retried), negative for an unrecoverable error (in which

case the integration is halted).

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

#### 4.6.9 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:

# IDALsPrecSetupFn

```
Definition
            typedef int (*IDALsPrecSetupFn)(realtype tt, N_Vector yy,
                                                N_{-}Vector yp, N_{-}Vector rr,
                                                realtype cj, void *user_data);
```

Purpose This function evaluates and/or preprocesses Jacobian-related data needed by the pre-

conditioner.

Arguments is the current value of the independent variable. tt

> is the current value of the dependent variable vector, y(t). уу

is the current value of  $\dot{y}(t)$ . ур

is the current value of the residual vector  $F(t, y, \dot{y})$ . rr

is the scalar in the system Jacobian, proportional to the inverse of the step сj

size ( $\alpha$  in Eq. (2.5)).

user\_data is a pointer to user data, the same as the user\_data parameter passed to the function IDASetUserData.

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

Notes

The operations performed by this function might include forming a crude approximate Jacobian, and performing an LU factorization on the resulting approximation.

With the default nonlinear solver (the native SUNDIALS Netwon method), each call to the preconditioner setup function is preceded by a call to the IDAResFn user function with the same (tt, yy, yp) 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  $\S 10.1.4$  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 §4.5.10.2. The unit roundoff can be accessed as UNIT\_ROUNDOFF defined in sundials\_types.h.

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

# 4.7 A parallel band-block-diagonal preconditioner module

A principal reason for using a parallel DAE solver such as IDA 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 (2.4) 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 [36] and is included in a software module within the IDA 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  $\bar{y}_m$  denote  $y_m$  and  $\dot{y}_m$  (respectively) augmented with those other components on which  $G_m$  depends. Then we have

$$G(t, y, \dot{y}) = [G_1(t, \bar{y}_1, \bar{y}_1), G_2(t, \bar{y}_2, \bar{y}_2), \dots, G_M(t, \bar{y}_M, \bar{y}_M)]^T,$$
(4.1)

and each of the blocks  $G_m(t, \bar{y}_m, \bar{y}_m)$  is uncoupled from the others.

The preconditioner associated with this decomposition has the form

$$P = diag[P_1, P_2, \dots, P_M] \tag{4.2}$$

where

$$P_m \approx \partial G_m / \partial y_m + \alpha \partial G_m / \partial \dot{y}_m \tag{4.3}$$

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  $\operatorname{mudq} + \operatorname{mldq} + 2$  evaluations of  $G_m$ , but only a matrix of bandwidth  $\operatorname{mukeep} + \operatorname{mlkeep} + 1$  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

$$Px = b (4.4)$$

reduces to solving each of the equations

$$P_m x_m = b_m \tag{4.5}$$

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.

The IDABBDPRE module calls two user-provided functions to construct P: a required function  $\operatorname{Gres}$  (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  $\operatorname{Gcomm}$  (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  $\operatorname{res}$ . Both functions take as input the same pointer  $\operatorname{user-data}$  as passed by the user to IDASetUserData and passed to the user's function  $\operatorname{res}$ . The user is responsible for providing space (presumably within  $\operatorname{user-data}$ ) for components of  $\operatorname{yy}$  and  $\operatorname{yp}$  that are communicated by  $\operatorname{Gcomm}$  from the other processors, and that are then used by  $\operatorname{Gres}$ , which should not do any communication.

# IDABBDLocalFn

Definition typedef int (\*IDABBDLocalFn)(sunindextype Nlocal, realtype tt, N\_Vector yy, N\_Vector yp, N\_Vector gval, void \*user\_data):

Purpose 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.

# IDABBDCommFn

Definition typedef int (\*IDABBDCommFn)(sunindextype Nlocal, realtype tt, N\_Vector yy, N\_Vector yp, void \*user\_data);

Purpose 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.

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 (tt, yy, yp) 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 (see below).

Besides the header files required for the integration of the DAE problem (see §4.3), 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 unchanged from the user main program presented in §4.4 are graved-out.

- 1. Initialize MPI
- 2. Set problem dimensions etc.
- 3. Set vectors of initial values
- 4. Create IDA object
- 5. Initialize IDA solver
- 6. Specify integration tolerances

# 7. Create linear solver object

When creating the iterative linear solver object, specify the use of left preconditioning (PREC\_LEFT) as IDA only supports left preconditioning.

- 8. Set linear solver optional inputs
- 9. Attach linear solver module
- 10. Set optional inputs

Note that the user should not overwrite the preconditioner setup function or solve function through calls to idIDASetPreconditioner optional input function.

# 11. Initialize the IDABBDPRE preconditioner module

Specify the upper and lower bandwidths mudq, mldq and mukeep, mlkeep and call

to allocate memory and initialize the internal preconditioner data. The last two arguments of IDABBDPrecInit are the two user-supplied functions described above.

- 12. Create nonlinear solver object
- 13. Attach nonlinear solver module
- 14. Set nonlinear solver optional inputs
- 15. Correct initial values
- 16. Specify rootfinding problem

#### 17. Advance solution in time

# 18. Get optional outputs

Additional optional outputs associated with IDABBDPRE are available by way of two routines described below, IDABBDPrecGetWorkSpace and IDABBDPrecGetNumGfnEvals.

- 19. Deallocate memory for solution vectors
- 20. Free solver memory
- 21. Free nonlinear solver memory
- 22. Free linear solver memory
- 23. Finalize MPI

The user-callable functions that initialize (step 11 above) or re-initialize the IDABBDPRE preconditioner module are described next.

# IDABBDPrecInit

Description The function IDABBDPrecInit initializes and allocates (internal) memory for the ID-ABBDPRE preconditioner.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

Nlocal (sunindextype) local vector dimension.

mudq (sunindextype) upper half-bandwidth to be used in the difference-quotient

Jacobian approximation.

mldq (sunindextype) lower half-bandwidth to be used in the difference-quotient

Jacobian approximation.

mukeep (sunindextype) upper half-bandwidth of the retained banded approximate

Jacobian block.

mlkeep (sunindextype) lower half-bandwidth of the retained banded approximate

Jacobian block.

dq\_rel\_yy (realtype) 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 (IDABBDLocalFn) the C function which computes the local residual approx-

imation  $G(t, y, \dot{y})$ .

Gcomm (IDABBDCommFn) the optional C function which performs all inter-process

communication required for the computation of  $G(t, y, \dot{y})$ .

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

IDALS\_SUCCESS The call to IDABBDPrecInit 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.

#### F2003 Name FIDABBDPrecInit

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 IDA 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 SUNLINSOL module, and/or one or more of the corresponding IDASet\*\*\* functions, must also be made (in the proper order).

# IDABBDPrecReInit

Call flag = IDABBDPrecReInit(ida\_mem, mudq, mldq, dq\_rel\_yy);

Description The function IDABBDPrecReInit reinitializes the IDABBDPRE preconditioner.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

mudq (sunindextype) upper half-bandwidth to be used in the difference-quotient

Jacobian approximation.

mldq (sunindextype) lower half-bandwidth to be used in the difference-quotient

Jacobian approximation.

dq\_rel\_yy (realtype) the relative increment in components of y used in the difference quotient approximations. The default is  $dq_rel_yy = \sqrt{unit\ roundoff}$ , which

can be specified by passing  $dq_rel_yy = 0.0$ .

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

IDALS\_SUCCESS The call to IDABBDPrecReInit 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.

 $F2003 \; \mathrm{Name} \; \; \mathtt{FIDABBDPrecReInit}$ 

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

#### IDABBDPrecGetWorkSpace

Call flag = IDABBDPrecGetWorkSpace(ida\_mem, &lenrwBBDP, &leniwBBDP);

Description The function IDABBDPrecGetWorkSpace returns the local sizes of the IDABBDPRE real and integer workspaces.

and meeger workspaces.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

lenrwBBDP (long int) local number of real values in the IDABBDPRE workspace. leniwBBDP (long int) local number of integer values in the IDABBDPRE workspace.

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

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 SUN-LINSOL 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.

F2003 Name FIDABBDPrecGetWorkSpace

# IDABBDPrecGetNumGfnEvals

Call flag = IDABBDPrecGetNumGfnEvals(ida\_mem, &ngevalsBBDP);

Description The function IDABBDPrecGetNumGfnEvals returns the cumulative number of calls to

the user  ${\tt Gres}$  function due to the finite difference approximation of the Jacobian blocks

used within IDABBDPRE's preconditioner setup function.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

ngevalsBBDP (long int) the cumulative number of calls to the user Gres function.

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

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.

F2003 Name FIDABBDPrecGetNumGfnEvals

In addition to the ngevalsBBDP Gres evaluations, the costs associated with IDABBDPRE also include nlinsetups LU factorizations, nlinsetups calls to Gcomm, npsolves banded backsolve calls, and nrevalsLS residual function evaluations, where nlinsetups is an optional IDA output (see §4.5.10.2), and npsolves and nrevalsLS are linear solver optional outputs (see §4.5.10.5).

# Chapter 5

# Using IDA for Fortran Applications

A Fortran 2003 module (fida\_mod) as well as a Fortran 77 style interface (FIDA) are provided to support the use of IDA, for the solution of DAE systems in a mixed Fortran/C setting. While IDA is written in C, it is assumed here that the user's calling program and user-supplied problem-defining routines are written in Fortran.

# 5.1 IDA Fortran 2003 Interface Module

The fida\_mod Fortran module defines interfaces to most IDA C functions using the intrinsic iso\_c\_binding module which provides a standardized mechanism for interoperating with C. All interfaced functions are named after the corresponding C function, but with a leading 'F'. For example, the IDA function IDACreate is interfaced as FIDACreate. Thus, the steps to use IDA and the function calls in Fortran 2003 are identical (ignoring language differences) to those in C. The C functions with Fortran 2003 interfaces indicate this in their description in Chapter 4. The Fortran 2003 IDA interface module can be accessed by the use statement, i.e. use fida\_mod, and linking to the library libsundials\_fida\_mod.lib in addition to libsundials\_ida.lib.

The Fortran 2003 interface modules were generated with SWIG Fortran, a fork of SWIG [37]. Users who are interested in the SWIG code used in the generation process should contact the SUNDIALS development team.

# 5.1.1 SUNDIALS Fortran 2003 Interface Modules

All of the generic SUNDIALS modules provide Fortran 2003 interface modules. Many of the generic module implementations provide Fortran 2003 interfaces (a complete list of modules with Fortran 2003 interfaces is given in Table 5.1). A module can be accessed with the use statement, e.g. use fnvector\_openmp\_mod, and linking to the Fortran 2003 library in addition to the C library, e.g. libsundials\_fnvecpenmp\_mod.lib and libsundials\_nvecopenmp.lib.

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

Further information on the Fortran 2003 interfaces specific to modules is given in the NVECTOR, SUNMATRIX, SUNLINSOL, and SUNNONLINSOL alongside the C documentation (chapters 7, 8, 9, and 10 respectively). For details on where the Fortran 2003 module (.mod) files and libraries are installed see Appendix A.

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

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

# 5.1.2 Data Types

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



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

# 5.1.3 Notable Fortran/C usage differences

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

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	<pre>integer(c_int), dimension(*)</pre>	
int*	return	<pre>integer(c_int), pointer, dimension(:)</pre>	
long*	in, inout, out	<pre>integer(c_long), dimension(*)</pre>	
long*	return	<pre>integer(c_long), pointer, dimension(:)</pre>	
realtype*	in, inout, out	real(c_double), dimension(*)	
realtype*	return	real(c_double), pointer, dimension(:)	
sunindextype*	in, inout, out	<pre>integer(c_long), dimension(*)</pre>	
sunindextype*	return	<pre>integer(c_long), pointer, dimension(:)</pre>	
realtype[]	in, inout, out	real(c_double), dimension(*)	
sunindextype[]	in, inout, out	<pre>integer(c_long), dimension(*)</pre>	
N_Vector	in, inout, out	$type(N_Vector)$	
N_Vector	return	$type(N_Vector)$ , pointer	
SUNMatrix	in, inout, out	type(SUNMatrix)	
SUNMatrix	return	type(SUNMatrix), pointer	
SUNLinearSolver	in, inout, out	type(SUNLinearSolver)	
SUNLinearSolver	return	type(SUNLinearSolver), pointer	
SUNNonlinearSolver	in, inout, out	type(SUNNonlinearSolver)	
SUNNonlinearSolver	return	type(SUNNonlinearSolver), pointer	
FILE*	in, inout, out, return	type(c_ptr)	
void*	in, inout, out, return	type(c_ptr)	
T**	in, inout, out, return	type(c_ptr)	
T***	in, inout, out, return	type(c_ptr)	
T****	in, inout, out, return	type(c_ptr)	

Table 5.2: C/Fortran 2003 Equivalent Types

# 5.1.3.1 Creating generic SUNDIALS objects

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

```
N_Vector x;
x = N_VNew_Serial(N);
Fortran code:
type(N_Vector), pointer :: x
x => FN_VNew_Serial(N)
```

C code:

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

# 5.1.3.2 Arrays and pointers

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

```
C \ code:
N_Vector x
realtype* xdata;
long int leniw, lenrw;
x = N_VNew_Serial(N);
/* capturing a returned array/pointer */
xdata = N_VGetArrayPointer(x)
/* passing array/pointer to a function */
N_VSetArrayPointer(xdata, x)
/* pointers that are out-parameters */
N_VSpace(x, &leniw, &lenrw);
Fortran code:
type(N_Vector), pointer :: x
real(c_double), pointer :: xdataptr(:)
real(c_double)
                        :: xdata(N)
integer(c_long)
                        :: leniw(1), lenrw(1)
x => FN_VNew_Serial(x)
! capturing a returned array/pointer
xdataptr => FN_VGetArrayPointer(x)
! passing array/pointer to a function
call FN_VSetArrayPointer(xdata, x)
! pointers that are out-parameters
call FN_VSpace(x, leniw, lenrw)
```

# 5.1.3.3 Passing procedure pointers and user data

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

Typically when passing user data to a SUNDIALS function, a user may simply cast some custom data structure as a void\*. When using the Fortran 2003 interfaces, the same thing can be achieved. Note, the custom data structure *does not* have to be bind(C) since it is never accessed on the C side.

 $C \ code$ :

```
MyUserData* udata;
```

```
void *cvode_mem;
ierr = CVodeSetUserData(cvode_mem, udata);
Fortran code:
type(MyUserData) :: udata
type(c_ptr) :: cvode_mem
ierr = FCVodeSetUserData(cvode_mem, c_loc(udata))
```

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

# 5.1.3.4 Passing NULL to optional parameters

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

```
SUNLinearSolver LS;
N_Vector x, b;

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

Fortran code:

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

A => null()

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

# 5.1.3.5 Working with N\_Vector arrays

Arrays of N\_Vector objects are interfaced to Fortran 2003 as opaque type(c\_ptr). As such, it is not possible to directly index an array of N\_Vector objects returned by the N\_Vector "VectorArray" operations, or packages with sensitivity capabilities. Instead, SUNDIALS provides a utility function FN\_VGetVecAtIndexVectorArray that can be called for accessing a vector in a vector array. The example below demonstrates this:

C code:

C code:

```
N_Vector x;
N_Vector* vecs;

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

Fortran code:

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

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

SUNDIALS also provides the functions FN\_VSetVecAtIndexVectorArray and FN\_VNewVectorArray for working with N\_Vector arrays. These functions are particularly useful for users of the Fortran interface to the NVECTOR\_MANYVECTOR or NVECTOR\_MPIMANYVECTOR when creating the subvector array. Both of these functions along with FN\_VGetVecAtIndexVectorArray are further described in Chapter 7.1.6.

# 5.1.3.6 Providing file pointers

FSUNDIALSFileOpen

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

```
Call fp = FSUNDIALSFileOpen(filename, mode)

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

Arguments filename (character(kind=C_CHAR, len=*)) - the path to the file to open
```

mode (character(kind=C\_CHAR, len=\*)) - the mode string given to fopen It should begin with one of the following characters:

"r" - open text file for reading

"r+" - open text file for reading and writing

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

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

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

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

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

# FSUNDIALSFileClose

Call call FSUNDIALSFileClose(fp)

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

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

Return value None

# 5.1.4 Important notes on portability

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

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

# 5.2 FIDA, an Interface Module for FORTRAN Applications

The fida interface module is a package of C functions which support the use of the IDA solver, for the solution of DAE systems, in a mixed FORTRAN/C setting. While IDA is written in C, it is assumed here that the user's calling program and user-supplied problem-defining routines are written in FORTRAN. This package provides the necessary interface to IDA for all supplied serial and parallel NVECTOR implementations.

# 5.3 Important note on portability

In this package, the names of the interface functions, and the names of the FORTRAN user routines called by them, appear as dummy names which are mapped to actual values by a series of definitions in the header files. By default, those mapping definitions depend in turn on the C macro F77\_FUNC defined in the header file sundials\_config.h. The mapping defined by F77\_FUNC in turn transforms the C interface names to match the name-mangling approach used by the supplied Fortran compiler.

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

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

# 5.4 Fortran Data Types

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

Integers: While SUNDIALS uses the configurable sunindextype type as the integer type for vector and matrix indices for its C code, the FORTRAN interfaces are more restricted. The sunindextype is only used for index values and pointers when filling sparse matrices. As for C, the sunindextype can be configured to be a 32- or 64-bit signed integer by setting the variable SUNDIALS\_INDEX\_TYPE at compile time (See Appendix A). The default value is int64\_t. A FORTRAN user should set this variable based on the integer type used for vector and matrix indices in their FORTRAN code. The corresponding FORTRAN types are:

- int32\_t equivalent to an INTEGER or INTEGER\*4 in FORTRAN
- int64\_t equivalent to an INTEGER\*8 in FORTRAN

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

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

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

# 5.4.1 FIDA routines

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

- Interface to the NVECTOR modules
  - FNVINITS (defined by NVECTOR\_SERIAL) interfaces to N\_VNewEmpty\_Serial.
  - FNVINITP (defined by NVECTOR\_PARALLEL) interfaces to N\_VNewEmpty\_Parallel.
  - FNVINITOMP (defined by NVECTOR\_OPENMP) interfaces to N\_VNewEmpty\_OpenMP.
  - FNVINITPTS (defined by NVECTOR\_PTHREADS) interfaces to N\_VNewEmpty\_Pthreads.
- Interface to the SUNMATRIX modules
  - FSUNBANDMATINIT (defined by SUNMATRIX\_BAND) interfaces to SUNBandMatrix.
  - FSUNDENSEMATINIT (defined by SUNMATRIX\_DENSE) interfaces to SUNDenseMatrix.
  - FSUNSPARSEMATINIT (defined by SUNMATRIX\_SPARSE) interfaces to SUNSparseMatrix.
- Interface to the SUNLINSOL modules
  - FSUNBANDLINSOLINIT (defined by SUNLINSOL\_BAND) interfaces to SUNLinSol\_Band.
  - FSUNDENSELINSOLINIT (defined by SUNLINSOL\_DENSE) interfaces to SUNLinSol\_Dense.
  - FSUNKLUINIT (defined by SUNLINSOL\_KLU) interfaces to SUNLinSol\_KLU.
  - FSUNKLUREINIT (defined by SUNLINSOL\_KLU) interfaces to SUNLinSol\_KLUReinit.
  - FSUNLAPACKBANDINIT (defined by SUNLINSOL\_LAPACKBAND) interfaces to SUNLinSol\_LapackBand.
  - FSUNLAPACKDENSEINIT (defined by SUNLINSOL\_LAPACKDENSE) interfaces to SUNLinSol\_LapackDense.
  - FSUNPCGINIT (defined by SUNLINSOL\_PCG) interfaces to SUNLinSol\_PCG.
  - FSUNSPBCGSINIT (defined by SUNLINSOL\_SPBCGS) interfaces to SUNLinSol\_SPBCGS.
  - FSUNSPFGMRINIT (defined by SUNLINSOL\_SPFGMR) interfaces to SUNLinSol\_SPFGMR.
  - FSUNSPGMRINIT (defined by SUNLINSOL\_SPGMR) interfaces to SUNLinSol\_SPGMR.
  - FSUNSPTFQMRINIT (defined by SUNLINSOL\_SPTFQMR) interfaces to SUNLinSol\_SPTFQMR.
  - FSUNSUPERLUMTINIT (defined by SUNLINSOL\_SUPERLUMT) interfaces to SUNLinSol\_SuperLUMT.

•

- Interface to the main IDA module
  - FIDAMALLOC interfaces to IDACreate, IDASetUserData, IDAInit, IDASStolerances, and IDASVtolerances.
  - FIDAREINIT interfaces to IDAReInit and IDASStolerances/IDASVtolerances.
  - FIDASETIIN, FIDASETVIN, and FIDASETRIN interface to IDASet\* functions.
  - FIDATOLREINIT interfaces to IDASStolerances/IDASVtolerances.
  - FIDACALCIC interfaces to IDACalcIC.
  - FIDAEWTSET interfaces to IDAWFtolerances.
  - FIDASOLVE interfaces to IDASolve, IDAGet\* functions, and to the optional output functions for the selected linear solver module.
  - FIDAGETDKY interfaces to IDAGetDky.
  - FIDAGETERRWEIGHTS interfaces to IDAGetErrWeights.
  - FIDAGETESTLOCALERR interfaces to IDAGetEstLocalErrors.
  - FIDAFREE interfaces to IDAFree.
- Interface to the IDALS module
  - FIDALSINIT interfaces to IDASetLinearSolver.
  - FIDALSSETEPSLIN interfaces to IDASetEpsLin
  - FIDALSSETJAC interfaces to IDASetJacTimes.
  - FIDALSSETPREC interfaces to IDASetPreconditioner.
  - FIDADENSESETJAC interfaces to IDASetJacFn.
  - FIDABANDSETJAC interfaces to IDASetJacFn.
  - FIDASPARSESETJAC interfaces to IDASetJacFn.

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

FIDA routine	IDA function	IDA type of
(Fortran, user-supplied)	(C, interface)	interface function
FIDARESFUN	FIDAresfn	IDAResFn
FIDAEWT	FIDAEwtSet	IDAEwtFn
FIDADJAC	FIDADenseJac	IDALsJacFn
FIDABJAC	FIDABandJac	IDALsJacFn
FIDASPJAC	FIDASparseJac	IDALsJacFn
FIDAPSOL	FIDAPSol	IDALsPrecSolveFn
FIDAPSET	FIDAPSet	IDALsPrecSetupFn
FIDAJTIMES	FIDAJtimes	IDALsJacTimesVecFn
FIDAJTSETUP	FIDAJTSetup	IDALsJacTimesSetupFn

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

# 5.4.2 Usage of the FIDA interface module

The usage of FIDA requires calls to a variety of interface functions, depending on the method options selected, and one or more user-supplied routines which define the problem to be solved. These function calls and user routines are summarized separately below. Some details are omitted, and the user is referred to the description of the corresponding IDA functions for information on the arguments of any given user-callable interface routine, or of a given user-supplied function called by an interface function. The usage of FIDA for rootfinding, and usage of FIDA with preconditioner modules, are each described in later sections.

# 1. Residual function specification

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

```
SUBROUTINE FIDARESFUN (T, Y, YP, R, IPAR, RPAR, IER) DIMENSION Y(*), YP(*), R(*), IPAR(*), RPAR(*)
```

It must set the R array to  $F(t, y, \dot{y})$ , the residual function of the DAE system, as a function of T = t and the arrays Y = y and YP =  $\dot{y}$ . The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC. It should return IER = 0 if it was successful, IER = 1 if it had a recoverable failure, or IER = -1 if it had a non-recoverable failure.

#### 2. NVECTOR module initialization

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

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

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

# 3. SUNMATRIX module initialization

In the case of a stiff system, the implicit BDF method involves the solution of linear systems related to the Jacobian of the DAE system. If using a Newton iteration with the direct SUNLINSOL linear solver module and one of the SUNMATRIX modules supplied with SUNDIALS, the user must make a call of the form

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

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

# 4. SUNLINSOL module initialization

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

```
CALL FSUNBANDLINSOLINIT(...)
CALL FSUNCHINIT(...)
CALL FSUNKLUINIT(...)
CALL FSUNLAPACKBANDINIT(...)
CALL FSUNLAPACKDENSEINIT(...)
CALL FSUNPCGINIT(...)
CALL FSUNSPBCGSINIT(...)
CALL FSUNSPFGMRINIT(...)
CALL FSUNSPFGMRINIT(...)
```

```
CALL FSUNSPTFQMRINIT(...)
CALL FSUNSUPERLUMTINIT(...)
```

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

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

```
CALL FSUNKLUSETORDERING(...)

CALL FSUNSUPERLUMTSETORDERING(...)

CALL FSUNPCGSETPRECTYPE(...)

CALL FSUNSPBCGSSETPRECTYPE(...)

CALL FSUNSPBCGSSETMAXL(...)

CALL FSUNSPFGMRSETGSTYPE(...)

CALL FSUNSPFGMRSETPRECTYPE(...)

CALL FSUNSPGMRSETGSTYPE(...)

CALL FSUNSPGMRSETPRECTYPE(...)

CALL FSUNSPGMRSETPRECTYPE(...)

CALL FSUNSPTFQMRSETPRECTYPE(...)

CALL FSUNSPTFQMRSETPRECTYPE(...)
```

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

#### 5. Problem specification

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

#### FIDAMALLOC

```
CALL FIDAMALLOC(TO, YO, YPO, IATOL, RTOL, ATOL,
Call
                                  IOUT, ROUT, IPAR, RPAR, IER)
Description
              This function provides required problem and solution specifications, specifies op-
              tional inputs, allocates internal memory, and initializes IDA.
Arguments
                     is the initial value of t.
              T0
              Y0
                     is an array of initial conditions for y.
              YP0
                     is an array of initial conditions for \dot{y}.
              IATOL specifies the type for absolute tolerance ATOL: 1 for scalar or 2 for array. If
                     IATOL = 3, the arguments RTOL and ATOL are ignored and the user is expected
                     to subsequently call FIDAEWTSET and provide the function FIDAEWT.
              RTOL
                     is the relative tolerance (scalar).
                     is the absolute tolerance (scalar or array).
                     is an integer array of length at least 21 for integer optional outputs.
                     is a real array of length at least 6 for real optional outputs.
              ROUT
                     is an integer array of user data which will be passed unmodified to all user-
                     provided routines.
                    is a real array of user data which will be passed unmodified to all user-
              RPAR
                     provided routines.
```

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

Notes The user integer data arrays IOUT and IPAR must be declared as INTEGER\*4 or INTEGER\*8 according to the C type long int.

Modifications to the user data arrays IPAR and RPAR inside a user-provided routine will be propagated to all subsequent calls to such routines.

The optional outputs associated with the main IDA integrator are listed in Table 5.4.

As an alternative to providing tolerances in the call to FIDAMALLOC, the user may provide a routine to compute the error weights used in the WRMS norm evaluations. If supplied, it must have the following form:

```
SUBROUTINE FIDAEWT (Y, EWT, IPAR, RPAR, IER)
DIMENSION Y(*), EWT(*), IPAR(*), RPAR(*)
```

It must set the positive components of the error weight vector EWT for the calculation of the WRMS norm of Y. On return, set IER = 0 if FIDAEWT was successful, and nonzero otherwise. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC.

If the FIDAEWT routine is provided, then, following the call to FIDAMALLOC, the user must make the call:

```
CALL FIDAEWTSET (FLAG, IER)
```

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

#### 6. Set optional inputs

Call FIDASETIIN, FIDASETRIN, and/or FIDASETVIN to set desired optional inputs, if any. See §5.5 for details.

#### 7. Linear solver interface specification

The variable-order, variable-coefficient BDF method used by IDA involves the solution of linear systems related to the system Jacobian  $J = \partial F/\partial y + \alpha \partial F/\partial \dot{y}$ . See Eq. (2.4). To attach the linear solver (and optionally the matrix) objects initialized in steps 3 and 4 above, the user of FIDA must initialize the IDALS linear solver interface. To attach any SUNLINSOL object (and optional SUNMATRIX object) to IDA, then following calls to initialize the SUNLINSOL (and SUNMATRIX) object(s) in steps 3 and 4 above, the user must make the call:

```
CALL FIDALSINIT(IER)
```

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

The previous routines FIDADLSINIT and FIDASPILSINIT 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.

#### IDALS with dense Jacobian matrix

As an option when using the IDALS interface with the SUNLINSOL\_DENSE or SUNLINSOL\_LAPACKDENSE linear solvers, the user may supply a routine that computes a dense approximation of the system Jacobian J. If supplied, it must have the following form:

```
SUBROUTINE FIDADJAC (NEQ, T, Y, YP, R, DJAC, CJ, EWT, H, IPAR, RPAR, WK1, WK2, WK3, IER)

DIMENSION Y(*), YP(*), R(*), EWT(*), DJAC(NEQ,*),

IPAR(*), RPAR(*), WK1(*), WK2(*), WK3(*)
```

This routine must compute the Jacobian and store it columnwise in DJAC. The vectors WK1, WK2, and WK3 of length NEQ are provided as work space for use in FIDADJAC. The input arguments T, Y, YP, R, and CJ are the current values of t, y,  $\dot{y}$ ,  $F(t, y, \dot{y})$ , and  $\alpha$ , respectively. The arrays IPAR (of

integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC. NOTE: The argument NEQ has a type consistent with C type long int even in the case when the LAPACK dense solver is to be used.

If the user's FIDADJAC uses difference quotient approximations, it may need to use the error weight array EWT and current stepsize H in the calculation of suitable increments. It may also need the unit roundoff, which can be obtained as the optional output ROUT(6), passed from the calling program to this routine using COMMON.

If the FIDADJAC routine is provided, then, following the call to FIDALSINIT the user must make the call:

```
CALL FIDADENSESETJAC (FLAG, IER)
```

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

#### IDALS with band Jacobian matrix

As an option when using the IDALs interface with the SUNLINSOL\_BAND or SUNLINSOL\_LAPACKBAND linear solvers, the user may supply a routine that computes a band approximation of the system Jacobian J. If supplied, it must have the following form:

```
SUBROUTINE FIDABJAC(NEQ, MU, ML, MDIM, T, Y, YP, R, CJ, BJAC, & EWT, H, IPAR, RPAR, WK1, WK2, WK3, IER)

DIMENSION Y(*), YP(*), R(*), EWT(*), BJAC(MDIM,*),

& IPAR(*), RPAR(*), WK1(*), WK2(*), WK3(*)
```

This routine must load the MDIM by NEQ array BJAC with the Jacobian matrix at the current  $(t,y,\dot{y})$  in band form. Store in BJAC(k,j) the Jacobian element  $J_{i,j}$  with k=i-j+MU+1  $(k=1\cdots$ ML+ MU+1) and  $j=1\cdots N$ . The vectors WK1, WK2, and WK3 of length NEQ are provided as work space for use in FIDABJAC. The input arguments T, Y, YP, R, and CJ are the current values of  $t,y,\dot{y},F(t,y,\dot{y})$ , and  $\alpha$ , respectively. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC. NOTE: The arguments NEQ, MU, ML, and MDIM have a type consistent with C type long int even in the case when the LAPACK band solver is to be used.

If the user's FIDABJAC uses difference quotient approximations, it may need to use the error weight array EWT and current stepsize H in the calculation of suitable increments. It may also need the unit roundoff, which can be obtained as the optional output ROUT(6), passed from the calling program to this routine using COMMON.

If the FIDABJAC routine is provided, then, following the call to FIDALSINIT, the user must make the call:

```
CALL FIDABANDSETJAC (FLAG, IER)
```

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

# ${\tt IDALS} \ \mathbf{with} \ \mathbf{sparse} \ \mathbf{Jacobian} \ \mathbf{matrix}$

When using the IDALS interface with the SUNLINSOL\_KLU or SUNLINSOL\_SUPERLUMT linear solvers, the user must supply the FIDASPJAC routine that computes a compressed-sparse-column (CSC) or compressed-sparse-row (CSR) approximation of the system Jacobian  $J = \partial F/\partial y + c_j \partial F/\partial \dot{y}$ . If supplied, it must have the following form:

```
SUBROUTINE FIDASPJAC(T, CJ, Y, YP, R, N, NNZ, JDATA, JINDEXVALS, & JINDEXPTRS, H, IPAR, RPAR, WK1, WK2, WK3, IER)
```

It must load the N by N compressed sparse column [or compressed sparse row] matrix with storage for NNZ nonzeros, stored in the arrays JDATA (nonzero values), JINDEXVALS (row [or column] indices for each nonzero), JINDEXPTRS (indices for start of each column [or row]), with the Jacobian matrix at the current (t,y) in CSC [or CSR] form (see summatrix\_sparse.h for more information). The arguments are T, the current time; CJ, scalar in the system proportional to the inverse step size; Y, an array containing state variables; YP, an array containing state derivatives; R, an array containing the system nonlinear residual; N, the number of matrix rows/columns in the Jacobian; NNZ, allocated length of nonzero storage; JDATA, nonzero values in the Jacobian (of length NNZ); JINDEXVALS, row [or column] indices for each nonzero in Jacobian (of length NNZ); JINDEXPTRS, pointers to each Jacobian column [or row] in the two preceding arrays (of length N+1); H, the current step size; IPAR, an array containing integer user data that was passed to FIDAMALLOC; RPAR, an array containing real user data that was passed to FIDAMALLOC; WK\*, work arrays containing temporary workspace of same size as Y; and IER, error return code (0 if successful, > 0 if a recoverable error occurred, or < 0 if an unrecoverable error occurred.)

To indicate that the FIDASPJAC routine has been provided, then following the call to FIDALSINIT, the following call must be made

```
CALL FIDASPARSESETJAC (IER)
```

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

#### IDALS with Jacobian-vector product

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

```
SUBROUTINE FIDAJTIMES(T, Y, YP, R, V, FJV, CJ, EWT, H, & IPAR, RPAR, WK1, WK2, IER)
DIMENSION Y(*), YP(*), R(*), V(*), FJV(*), EWT(*), & IPAR(*), RPAR(*), WK1(*), WK2(*)
```

This routine must compute the product vector Jv, where the vector v is stored in V, and store the product in FJV. On return, set IER = 0 if FIDAJTIMES was successful, and nonzero otherwise. The vectors W1K and WK2, of length NEQ, are provided as work space for use in FIDAJTIMES. The input arguments T, Y, YP, R, and CJ are the current values of t, y,  $\dot{y}$ ,  $F(t, y, \dot{y})$ , and  $\alpha$ , respectively. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC.

If the user's FIDAJTIMES uses difference quotient approximations, it may need to use the error weight array EWT and current stepsize H in the calculation of suitable increments. It may also need the unit roundoff, which can be obtained as the optional output ROUT(6), passed from the calling program to this routine using COMMON.

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

```
SUBROUTINE FIDAJTSETUP (T, Y, YP, R, CJ, EWT, H, IPAR, RPAR, IER) DIMENSION Y(*), YP(*), R(*), EWT(*), IPAR(*), RPAR(*)
```

Typically this routine will use only T, Y, and IDAYP. It should compute any necessary data for subsequent calls to FIDAJTIMES. On return, set IER = 0 if FIDAJTSETUP was successful, and nonzero otherwise. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC.

To indicate that the FIDAJTIMES and FIDAJTSETUP routines have been provided, then following the call to FIDALSINIT, the following call must be made

```
CALL FIDALSSETJAC (FLAG, IER)
```

with  $FLAG \neq 0$ . The return flag IER is 0 if successful, or negative if a memory error occurred.

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

If the user calls FIDALSSETJAC, the routine FIDAJTSETUP must be provided, even if it is not needed, and it must return IER=0.

#### IDALS with preconditioning

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

```
SUBROUTINE FIDAPSOL(T, Y, YP, R, RV, ZV, CJ, DELTA, EWT, & IPAR, RPAR, IER)

DIMENSION Y(*), YP(*), R(*), RV(*), ZV(*), EWT(*), & IPAR(*), RPAR(*)
```

It must solve the preconditioner linear system Pz=r, where  $r=\mathrm{RV}$  is input, and store the solution z in ZV. Here P is the left preconditioner. The input arguments T, Y, YP, R, and CJ are the current values of  $t, y, \dot{y}, F(t, y, \dot{y})$ , and  $\alpha$ , respectively. On return, set IER = 0 if FIDAPSOL was successful, set IER positive if a recoverable error occurred, and set IER negative if a non-recoverable error occurred.

The arguments EWT and DELTA are input and provide the error weight array and a scalar tolerance, respectively, for use by FIDAPSOL if it uses an iterative method in its solution. In that case, the residual vector  $\rho = r - Pz$  of the system should be made less than DELTA in weighted  $\ell_2$  norm, i.e.  $\sqrt{\sum (\rho_i * \text{EWT}[i])^2} < \text{DELTA}$ . The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC.

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

```
SUBROUTINE FIDAPSET(T, Y, YP, R, CJ, EWT, H, 
& IPAR, RPAR, IER)
DIMENSION Y(*), YP(*), R(*), EWT(*), 
& IPAR(*), RPAR(*)
```

It must perform any evaluation of Jacobian-related data and preprocessing needed for the solution of the preconditioner linear systems by FIDAPSOL. The input arguments T, Y, YP, R, and CJ are the current values of  $t, y, \dot{y}, F(t, y, \dot{y})$ , and  $\alpha$ , respectively. On return, set IER = 0 if FIDAPSET was successful, set IER positive if a recoverable error occurred, and set IER negative if a non-recoverable error occurred. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC.

If the user's FIDAPSET uses difference quotient approximations, it may need to use the error weight array EWT and current stepsize H in the calculation of suitable increments. It may also need the unit roundoff, which can be obtained as the optional output ROUT(6), passed from the calling program to this routine using COMMON.

To indicate that the FIDAPSET and FIDAPSOL routines are supplied, then following the call to FIDALSINIT, the user must call

with FLAG  $\neq 0$ . The return flag IER is 0 if successful, or negative if a memory error occurred. In addition, the user must supply preconditioner routines FIDAPSET and FIDAPSOL.

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

If the user calls FIDALSSETPREC, the subroutine FIDAPSET must be provided, even if it is not needed, and it must return IER = 0.

#### 8. Correct initial values

Optionally, to correct the initial values y and/or  $\dot{y}$ , make the call

(See §2.1 for details.) The arguments are as follows: ICOPT is 1 for initializing the algebraic components of y and differential components of  $\dot{y}$ , or 2 for initializing all of y. IER is an error return flag, which is 0 for success, or negative for a failure (see IDACalcIC return values).

#### 9. Problem solution

Carrying out the integration is accomplished by making calls as follows:

```
CALL FIDASOLVE (TOUT, T, Y, YP, ITASK, IER)
```

The arguments are as follows. TOUT specifies the next value of t at which a solution is desired (input). T is the value of t reached by the solver on output. Y is an array containing the computed solution vector y on output. YP is an array containing the computed solution vector  $\dot{y}$  on output. ITASK is a task indicator and should be set to 1 for normal mode (overshoot TOUT and interpolate), or to 2 for one-step mode (return after each internal step taken). IER is a completion flag and will be set to a positive value upon successful return or to a negative value if an error occurred. These values correspond to the IDASolve returns (see §4.5.7 and §B.2). The current values of the optional outputs are available in IOUT and ROUT (see Table 5.4).

#### 10. Additional solution output

After a successful return from FIDASOLVE, the routine FIDAGETDKY may be called to get interpolated values of y or any derivative  $d^k y/dt^k$  for k not exceeding the current method order, and for any value of t in the last internal step taken by IDA. The call is as follows:

```
CALL FIDAGETDKY (T, K, DKY, IER)
```

where T is the input value of t at which solution derivative is desired, K is the derivative order, and DKY is an array containing the computed vector  $y^{(K)}(t)$  on return. The value of T must lie between TCUR - HLAST and TCUR. The value of K must satisfy  $0 \le K \le \text{QLAST}$ . (See the optional outputs for TCUR, HLAST, and QLAST.) The return flag IER is set to 0 upon successful return, or to a negative value to indicate an illegal input.

#### 11. Problem reinitialization

To re-initialize the IDA solver for the solution of a new problem of the same size as one already solved, make the following call:

```
CALL FIDAREINIT (TO, YO, YPO, IATOL, RTOL, ATOL, IER)
```

The arguments have the same names and meanings as those of FIDAMALLOC. FIDAREINIT performs the same initializations as FIDAMALLOC, but does no memory allocation, using instead the existing internal memory created by the previous FIDAMALLOC call.



Following this call, if the choice of linear solver is being changed then a user must make a call to create the alternate SUNLINSOL module and then attach it to the IDALS interface, as shown above. If only linear solver parameters are being modified, then these calls may be made without re-attaching to the IDALS interface.

#### 12. Memory deallocation

To free the internal memory created by the call to FIDAMALLOC, FIDALSINIT, FNVINIT\* and FSUN\*\*\*MATINIT, make the call

CALL FIDAFREE

# 5.5 FIDA optional input and output

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

```
CALL FIDASETIIN(KEY, IVAL, IER)
CALL FIDASETRIN(KEY, RVAL, IER)
CALL FIDASETVIN(KEY, VVAL, IER)
```

where KEY is a quoted string indicating which optional input is set (see Table 5.3), IVAL is the input integer value, RVAL is the input real value (scalar), VVAL is the input real array, and IER is an integer return flag which is set to 0 on success and a negative value if a failure occurred. IVAL should be declared so as to match C type long int.

When using FIDASETVIN to specify the variable types (KEY = 'ID\_VEC') the components in the array VVAL must be 1.0 to indicate a differential variable, or 0.0 to indicate an algebraic variable. Note that this array is required only if FIDACALCIC is to be called with ICOPT = 1, or if algebraic variables are suppressed from the error test (indicated using FIDASETIIN with KEY = 'SUPPRESS\_ALG'). When using FIDASETVIN to specify optional constraints on the solution vector (KEY = 'CONSTR\_VEC') the components in the array VVAL should be one of -2.0, -1.0, 0.0, 1.0, or 2.0. See the description of IDASetConstraints (§4.5.8.1) for details.

The optional outputs from the IDA solver are accessed not through individual functions, but rather through a pair of arrays, IOUT (integer type) of dimension at least 21, and ROUT (real type) of dimension at least 6. These arrays are owned (and allocated) by the user and are passed as arguments to FIDAMALLOC. Table 5.4 lists the entries in these two arrays and specifies the optional variable as well as the IDA function which is actually called to extract the optional output.

For more details on the optional inputs and outputs, see §4.5.8 and §4.5.10.

In addition to the optional inputs communicated through FIDASET\* calls and the optional outputs extracted from IOUT and ROUT, the following user-callable routines are available:

To reset the tolerances at any time, make the following call:

```
CALL FIDATOLREINIT (IATOL, RTOL, ATOL, IER)
```

The tolerance arguments have the same names and meanings as those of FIDAMALLOC. The error return flag IER is 0 if successful, and negative if there was a memory failure or illegal input.

To obtain the error weight array EWT, containing the multiplicative error weights used the WRMS norms, make the following call:

```
CALL FIDAGETERRWEIGHTS (EWT, IER)
```

This computes the EWT array, normally defined by Eq. (2.6). The array EWT, of length NEQ or NLOCAL, must already have been declared by the user. The error return flag IER is zero if successful, and negative if there was a memory error.

Table 5.3: Keys for setting FIDA optional inputs

# Integer optional inputs (FIDASETIIN)

Key	Optional input	Default value
MAX_ORD	Maximum LMM method order	5
MAX_NSTEPS	Maximum no. of internal steps before $t_{\text{out}}$	500
MAX_ERRFAIL	Maximum no. of error test failures	10
MAX_NITERS	Maximum no. of nonlinear iterations	4
MAX_CONVFAIL	Maximum no. of convergence failures	10
SUPPRESS_ALG	Suppress alg. vars. from error test $(1 = SUNTRUE)$	0 (= SUNFALSE)
MAX_NSTEPS_IC	Maximum no. of steps for IC calc.	5
MAX_NITERS_IC	Maximum no. of Newton iterations for IC calc.	10
MAX_NJE_IC	Maximum no. of Jac. evals fo IC calc.	4
LS_OFF_IC	Turn off line search $(1 = SUNTRUE)$	0 (= SUNFALSE)

# Real optional inputs (FIDASETRIN)

Table (			
Key	Optional input	Default value	
INIT_STEP	Initial step size	estimated	
MAX_STEP	Maximum absolute step size	$\infty$	
STOP_TIME	Value of $t_{stop}$	undefined	
NLCONV_COEF	Coeff. in the nonlinear conv. test	0.33	
NLCONV_COEF_IC	Coeff. in the nonlinear conv. test for IC calc.	0.0033	
STEP_TOL_IC	Lower bound on Newton step for IC calc.	$uround^{2/3}$	

# Real vector optional inputs $({\tt FIDASETVIN})$

Key	Optional input	Default value
ID_VEC	Differential/algebraic component types	undefined
CONSTR_VEC	Inequality constraints on solution	undefined

Table 5.4: Description of the FIDA optional output arrays IOUT and ROUT Integer output array IOUT

integer output array 1001					
Index	Optional output	IDA function			
IDA main solver					
1	LENRW	IDAGetWorkSpace			
2	LENIW	IDAGetWorkSpace			
3	NST	IDAGetNumSteps			
4	NRE	IDAGetNumResEvals			
5	NETF	IDAGetNumErrTestFails			
6	NNCFAILS	IDAGetNonlinSolvConvFails			
7	NNI	IDAGetNumNonlinSolvIters			
8	NSETUPS	IDAGetNumLinSolvSetups			
9	QLAST	IDAGetLastOrder			
10	QCUR	IDAGetCurrentOrder			
11	NBCKTRKOPS	IDAGetNumBacktrackOps			
12	NGE	IDAGetNumGEvals			
	IDALS linear solver interface				
13	LENRWLS	IDAGetLinWorkSpace			
14	LENIWLS	IDAGetLinWorkSpace			
15	$\mathtt{LS}_{-}\mathtt{FLAG}$	IDAGetLastLinFlag			
16	NRELS	IDAGetNumLinResEvals			
17	NJE	IDAGetNumJacEvals			
18	NJTS	IDAGetNumJTSetupEvals			
19	NJT	IDAGetNumJtimesEvals			
20	NPE	IDAGetNumPrecEvals			
21	NPS	IDAGetNumPrecSolves			
22	NLI	IDAGetNumLinIters			
23	NCFL	IDAGetNumLinConvFails			

Real output array ROUT

recar sarpar array 1000				
Index	Optional output	IDA function		
1	HO_USED	IDAGetActualInitStep		
2	HLAST	IDAGetLastStep		
3	HCUR	IDAGetCurrentStep		
4	TCUR	IDAGetCurrentTime		
5	TOLFACT	IDAGetTolScaleFactor		
6	UROUND	unit roundoff		

To obtain the estimated local errors, following a successful call to FIDASOLVE, make the following call:

```
CALL FIDAGETESTLOCALERR (ELE, IER)
```

This computes the ELE array of estimated local errors as of the last step taken. The array ELE must already have been declared by the user. The error return flag IER is zero if successful, and negative if there was a memory error.

# 5.5.1 Usage of the FIDAROOT interface to rootfinding

The FIDAROOT interface package allows programs written in FORTRAN to use the rootfinding feature of the IDA solver module. The user-callable functions in FIDAROOT, with the corresponding IDA functions, are as follows:

- FIDAROOTINIT interfaces to IDARootInit.
- FIDAROOTINFO interfaces to IDAGetRootInfo.
- FIDAROOTFREE interfaces to IDARootFree.

Note that, at this time FIDAROOT does not provide support to specify the direction of zero-crossing that is to be monitored. Instead, all roots are considered. However, the actual direction of zero-crossing is reported (through the sign of the non-zero elements in the array INFO returned by FIDAROTINFO).

In order to use the rootfinding feature of IDA, the following call must be made, after calling FIDAMALLOC but prior to calling FIDASOLVE, to allocate and initialize memory for the FIDAROOT module:

```
CALL FIDAROOTINIT (NRTFN, IER)
```

The arguments are as follows: NRTFN is the number of root functions. IER is a return completion flag; its values are 0 for success, -1 if the IDA memory was NULL, and -14 if a memory allocation failed.

To specify the functions whose roots are to be found, the user must define the following routine:

```
SUBROUTINE FIDAROOTFN (T, Y, YP, G, IPAR, RPAR, IER)
DIMENSION Y(*), YP(*), G(*), IPAR(*), RPAR(*)
```

It must set the G array, of length NRTFN, with components  $g_i(t, y, \dot{y})$ , as a function of T = t and the arrays Y = y and  $YP = \dot{y}$ . The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC. Set IER on 0 if successful, or on a non-zero value if an error occurred

When making calls to FIDASOLVE to solve the DAE system, the occurrence of a root is flagged by the return value IER = 2. In that case, if NRTFN > 1, the functions  $g_i$  which were found to have a root can be identified by making the following call:

```
CALL FIDAROOTINFO (NRTFN, INFO, IER)
```

The arguments are as follows: NRTFN is the number of root functions. INFO is an integer array of length NRTFN with root information. IER is a return completion flag; its values are 0 for success, negative if there was a memory failure. The returned values of INFO(i) (i=1,...,NRTFN) are 0 or  $\pm 1$ , such that INFO(i) = +1 if  $g_i$  was found to have a root and  $g_i$  is increasing, INFO(i) = -1 if  $g_i$  was found to have a root and  $g_i$  is dereasing, and INFO(i) = 0 otherwise.

The total number of calls made to the root function FIDAROOTFN, denoted NGE, can be obtained from IOUT(12). If the FIDA/IDA memory block is reinitialized to solve a different problem via a call to FIDAREINIT, then the counter NGE is reset to zero.

To free the memory resources allocated by a prior call to FIDAROOTINIT, make the following call:

```
CALL FIDAROOTFREE
```

See §4.5.6 for additional information on the rootfinding feature.

# 5.5.2 Usage of the FIDABBD interface to IDABBDPRE

The FIDABBD interface sub-module is a package of C functions which, as part of the FIDA interface module, support the use of the IDA solver with the parallel NVECTOR\_PARALLEL module, in a combination of any of the Krylov iterative solver modules with the IDABBDPRE preconditioner module (see §4.7).

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

- FIDABBDINIT interfaces to IDABBDPrecAlloc.
- FIDABBDREINIT interfaces to IDABBDPrecReInit.
- FIDABBDOPT interfaces to IDABBDPRE optional output functions.
- FIDABBDFREE interfaces to IDABBDPrecFree.

In addition to the FORTRAN residual function FIDARESFUN, the user-supplied functions used by this package, are listed below, each with the corresponding interface function which calls it (and its type within IDABBDPRE or IDA):

FIDABBD routine (FORTRAN)	IDA function (C)	IDA function type
FIDAGLOCFN	FIDAgloc	IDABBDLocalFn
FIDACOMMFN	FIDAcfn	IDABBDCommFn
FIDAJTIMES	FIDAJtimes	IDALsJacTimesVecFn
FIDAJTSETUP	FIDAJTSetup	IDALsJacTimesSetupFn

As with the rest of the FIDA routines, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program. Additionally, based on flags discussed above in §5.4.1, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file fidabbd.h.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in §5.4.2 are graved-out.

- 1. Residual function specification
- 2. NVECTOR module initialization
- 3. SUNLINSOL module initialization

Initialize one of the iterative SUNLINSOL modules, by calling one of FSUNPCGINIT, FSUNSPBCGSINIT, FSUNSPFGMRINIT, FSUNSPGMRINIT or FSUNSPTFQMRINIT.

- 4. Problem specification
- 5. Set optional inputs
- 6. Linear solver interface specification

Initialize the IDALS iterative linear solver interface by calling FIDALSINIT.

#### 7. BBD preconditioner initialization

To initialize the IDABBDPRE preconditioner, make the following call:

```
CALL FIDABBDINIT (NLOCAL, MUDQ, MLDQ, MU, ML, DQRELY, IER)
```

The arguments are as follows. NLOCAL is the local size of vectors on this processor. MUDQ and MLDQ are the upper and lower half-bandwidths to be used in the computation of the local Jacobian blocks by difference quotients. These may be smaller than the true half-bandwidths of the Jacobian of the local block of G, when smaller values may provide greater efficiency. MU and ML are the upper

and lower half-bandwidths of the band matrix that is retained as an approximation of the local Jacobian block. These may be smaller than MUDQ and MLDQ. DQRELY is the relative increment factor in y for difference quotients (optional). A value of 0.0 indicates the default,  $\sqrt{\text{unit roundoff. IER}}$  is a return completion flag. A value of 0 indicates success, while a value of -1 indicates that a memory failure occurred or that an input had an illegal value.

- 8. Correct initial values
- 9. Problem solution
- 10. Additional solution output

# 11. IDABBDPRE Optional outputs

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

```
CALL FIDABBDOPT (LENRWBBD, LENIWBBD, NGEBBD)
```

The arguments should be consistent with C type long int. Their returned values are as follows: LENRWBBD is the length of real preconditioner work space, in realtype words. LENIWBBD is the length of integer preconditioner work space, in integer words. Both of these sizes are local to the current processor. NGEBBD is the number of  $G(t, y, \dot{y})$  evaluations (calls to FIDALOCFN) so far.

#### 12. Problem reinitialization

If a sequence of problems of the same size is being solved using the same linear solver in combination with the IDABBDPRE preconditioner, then the IDA package can be re-initialized for the second and subsequent problems by calling FIDAREINIT, following which a call to FIDABBDINIT may or may not be needed. If the input arguments are the same, no FIDABBDINIT call is needed. If there is a change in input arguments other than MU or ML, then the user program should make the call

```
CALL FIDABBDREINIT (NLOCAL, MUDQ, MLDQ, DQRELY, IER)
```

This reinitializes the IDABBDPRE preconditioner, but without reallocating its memory. The arguments of the FIDABBDREINIT routine have the same names and meanings as those of FIDABBDINIT. If the value of MU or ML is being changed, then a call to FIDABBDINIT must be made. Finally, if there is a change in any of the linear solver inputs, then a call to one of FSUN\*\*\*\*INIT, followed by a call to FIDALSINIT must also be made; in this case the linear solver memory is reallocated.

#### 13. Memory deallocation

(The memory allocated for the FIDABBD module is deallocated automatically by FIDAFREE.)

# 14. User-supplied routines

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

```
SUBROUTINE FIDAGLOCFN (NLOC, T, YLOC, YPLOC, GLOC, IPAR, RPAR, IER)
DIMENSION YLOC(*), YPLOC(*), GLOC(*), IPAR(*), RPAR(*)
```

This routine is to evaluate the function  $G(t,y,\dot{y})$  approximating F (possibly identical to F), in terms of T=t, and the arrays YLOC and YPLOC (of length NLOC), which are the sub-vectors of y and  $\dot{y}$  local to this processor. The resulting (local) sub-vector is to be stored in the array GLOC. IER is a return flag that should be set to 0 if successful, to 1 (for a recoverable error), or to -1 (for a non-recoverable error). The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC.

SUBROUTINE FIDACOMMFN (NLOC, T, YLOC, YPLOC, IPAR, RPAR, IER) DIMENSION YLOC(\*), YPLOC(\*), IPAR(\*), RPAR(\*)

This routine is to perform the inter-processor communication necessary for the FIDAGLOCFN routine. Each call to FIDACOMMFN is preceded by a call to the residual routine FIDARESFUN with the same arguments T, YLOC, and YPLOC. Thus FIDACOMMFN can omit any communications done by FIDARESFUN if relevant to the evaluation of GLOC. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FIDAMALLOC. IER is a return flag that should be set to 0 if successful, to 1 (for a recoverable error), or to -1 (for a non-recoverable error).

The subroutine FIDACOMMFN must be supplied even if it is empty, and it must return IER = 0.

Optionally, the user can supply routines FIDAJTIMES and FIDAJTSETUP for the evaluation of Jacobian-vector products, as described above in step 7 in §5.4.2.



# Chapter 6

# IDA Features for GPU Accelerated Computing

This chapter is concerned with using GPU-acceleration and IDA for the solution of DAEs.

# 6.1 SUNDIALS GPU Programming Model

In this section, we introduce the SUNDIALS GPU programming model and highlight SUNDIALS GPU features. The model leverages the fact that all of the SUNDIALS packages interact with simulation data either through the shared vector, matrix, and solver APIs (see §7, §8, §9, and §10) or through user-supplied callback functions. Thus, under the model, the overall structure of the user's calling program, and the way users interact with the SUNDIALS packages is similar to using SUNDIALS in CPU-only environments.

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

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

In addition, SUNDIALS provides the SUNMemoryHelper API §11.1 to support applications which

implement their own memory management or memory pooling.

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

Module	CUDA	m ROCm/HIP	oneAPI	Unmanaged memory	UVM
NVECTOR_CUDA (§7.9)	<b>√</b>			<b>√</b>	<b>√</b>
NVECTOR_RAJA (§7.11)	<b>√</b>			<b>√</b>	🗸
NVECTOR_OPENMPDEV (§7.13)	<b>√</b>	✓	<b>√</b>	<b>√</b>	
SUNMATRIX_CUSPARSE (§8.7)	<b>√</b>			<b>√</b>	<b>  √</b>
SUNLINSOL_CUSOLVERSP_BATCHQR (§9.12)	✓			✓	🗸
SUNLINSOL_SPGMR (§9.14)	†	†	†	†	†
SUNLINSOL_SPFGMR ( $\S9.15$ )	†	†	†	†	†
SUNLINSOL_SPTFQMR ( $\S9.17$ )	†	†	†	†	†
SUNLINSOL_SPBCGS (§9.16)	†	†	†	†	†
SUNLINSOL_PCG (§9.18)	†	†	†	†	†
SUNNONLINSOL_NEWTON (§10.3)	†	†	†	†	†
SUNNONLINSOL_FIXEDPOINT (§??)	†	†	†	†	†

# 6.2 Steps for Using GPU Accelerated SUNDIALS

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

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

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

# Chapter 7

# Description of the NVECTOR module

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

# 7.1 The NVECTOR API

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

#### 7.1.1 NVECTOR core functions

#### N\_VGetVectorID

Call id = N\_VGetVectorID(w);

Description Returns the vector type identifier for the vector w. It is used to determine the vector

implementation type (e.g. serial, parallel,...) from the abstract N\_Vector interface.

Arguments w (N\_Vector) a NVECTOR object

Return value This function returns an N-Vector\_ID. Possible values are given in Table 7.1.

F2003 Name FN\_VGetVectorID

N\_VClone

Call v = N\_VClone(w);

Description Creates a new N\_Vector of the same type as an existing vector w and sets the ops field.

It does not copy the vector, but rather allocates storage for the new vector.

Arguments w (N\_Vector) a NVECTOR object

Return value This function returns an N\_Vector object. If an error occurs, then this routine will

return NULL.

 $F2003\ Name\ FN\_VClone$ 

# $N_{V}$ CloneEmpty

Call v = N\_VCloneEmpty(w);

Description Creates a new N\_Vector of the same type as an existing vector w and sets the ops field.

It does not allocate storage for data.

Arguments w (N\_Vector) a NVECTOR object

Return value This function returns an N\_Vector object. If an error occurs, then this routine will

return NULL.

F2003 Name FN\_VCloneEmpty

#### N\_VDestroy

Call N\_VDestroy(v);

Description Destroys the N\_Vector v and frees memory allocated for its internal data.

Arguments v (N\_Vector) a NVECTOR object to destroy

Return value None

F2003 Name FN\_VDestroy

#### N\_VSpace

Call N\_VSpace(v, &lrw, &liw);

Description Returns storage requirements for one N\_Vector. 1rw contains the number of realtype

words and liw contains the number of integer words, This function is advisory only, for use in determining a user's total space requirements; it could be a dummy function in

a user-supplied NVECTOR module if that information is not of interest.

Arguments v (N\_Vector) a NVECTOR object

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

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

Return value None

 $F2003\ Name\ FN_VSpace$ 

F2003 Call integer(c\_long) :: lrw(1), liw(1)

call FN\_VSpace\_Serial(v, lrw, liw)

# $N_{-}VGetArrayPointer$

Call vdata = N\_VGetArrayPointer(v);

Description Returns a pointer to a 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 CPU.

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

with SUNDIALS.

Arguments v (N\_Vector) a NVECTOR object

Return value realtype\*

F2003 Name FN\_VGetArrayPointer

#### N\_VGetDeviceArrayPointer

Call vdata = N\_VGetDeviceArrayPointer(v);

Description Returns a device pointer to a realtype array from the N\_Vector v. Note that this

assumes that the internal data in  ${\tt N\_Vector}$  is a contiguous array of  ${\tt realtype}$  and is

accessible from the device (e.g., GPU).

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

Arguments v (N\_Vector) a NVECTOR object

Return value realtype\*

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

SUNDIALS vectors will return NULL.

F2003 Name FN\_VGetDeviceArrayPointer

#### N\_VSetArrayPointer

Call N\_VSetArrayPointer(vdata, v);

Description Overwrites the pointer to the data in an N\_Vector with a given realtype\*. Note that

this assumes that the internal data in  $N_{-}$ Vector is a contiguous array of realtype. This routine is only used in the interfaces to the dense (serial) linear solver, hence need not

exist in a user-supplied NVECTOR module for a parallel environment.

Arguments v (N\_Vector) a NVECTOR object

Return value None

F2003 Name FN\_VSetArrayPointer

# N\_VGetCommunicator

Call N\_VGetCommunicator(v);

Description Returns a pointer to the MPI\_Comm object associated with the vector (if applicable). For

MPI-unaware vector implementations, this should return NULL.

Arguments v (N\_Vector) a NVECTOR object

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

 $F2003 \; \mathrm{Name} \; \; \mathtt{FN\_VGetCommunicator}$ 

#### N\_VGetLength

Call N\_VGetLength(v);

Description Returns the global length (number of 'active' entries) in the NVECTOR v. This value

should be cumulative across all processes if the vector is used in a parallel environment. If v contains additional storage, e.g., for parallel communication, those entries should

not be included.

Arguments v (N\_Vector) a NVECTOR object

Return value sunindextype F2003 Name FN\_VGetLength

#### $N_VLinearSum$

Call N\_VLinearSum(a, x, b, y, z);

Description Performs the operation z = ax + by, where a and b are realtype scalars and x and y

are of type N\_Vector:  $z_i = ax_i + by_i$ ,  $i = 0, \ldots, n-1$ .

Arguments a (realtype) constant that scales x

x (N\_Vector) a NVECTOR object

b (realtype) constant that scales y

y (N\_Vector) a NVECTOR object

z (N\_Vector) a NVECTOR object containing the result

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

 $F2003 \; Name \; FN_VLinearSum$ 

#### $N_{-}VConst$

Call N\_VConst(c, z);

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

Arguments c (realtype) constant to set all components of z to

z (N\_Vector) a NVECTOR object containing the result

Return value None

F2003 Name FN\_VConst

#### N\_VProd

Call N\_VProd(x, y, z);

Description Sets the N\_Vector z to be the component-wise product of the N\_Vector inputs x and y:

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

Arguments x (N\_Vector) a NVECTOR object

y (N\_Vector) a NVECTOR object

z (N\_Vector) a NVECTOR object containing the result

Return value None

F2003 Name FN\_VProd

#### N\_VDiv

Call N\_VDiv(x, y, z);

Description Sets the N\_Vector z to be the component-wise ratio of the N\_Vector inputs x and y:

 $z_i = x_i/y_i, i = 0, \dots, n-1$ . The  $y_i$  may not be tested for 0 values. It should only be

called with a y that is guaranteed to have all nonzero components.

Arguments x (N\_Vector) a NVECTOR object

y (N\_Vector) a NVECTOR object

z (N\_Vector) a NVECTOR object containing the result

Return value None

F2003 Name FN\_VDiv

 $N_{-}VScale$ 

Call N\_VScale(c, x, z);

Description Scales the N\_Vector x by the realtype scalar c and returns the result in z:  $z_i = cx_i$ , i =

 $0, \ldots, n-1.$ 

Arguments c (realtype) constant that scales the vector x

x (N\_Vector) a NVECTOR object

z (N\_Vector) a NVECTOR object containing the result

Return value None

F2003 Name FN\_VScale

 $N_{VAbs}$ 

Call N\_VAbs(x, z);

Description Sets the components of the N\_Vector z to be the absolute values of the components of

the N\_Vector x:  $z_i = |x_i|, i = 0, ..., n-1$ .

Arguments x (N\_Vector) a NVECTOR object

z (N\_Vector) a NVECTOR object containing the result

Return value None F2003 Name FN VAbs

N\_VInv

Call N\_VInv(x, z);

Description Sets the components of the N\_Vector z to be the inverses of the components of the

N-Vector x:  $z_i = 1.0/x_i$ , i = 0, ..., n-1. This routine may not check for division by 0. It should be called only with an x which is guaranteed to have all nonzero components.

Arguments x (N\_Vector) a NVECTOR object to

z (N\_Vector) a NVECTOR object containing the result

Return value None F2003 Name FN\_VInv

N\_VAddConst

Call N\_VAddConst(x, b, z);

Description Adds the realtype scalar b to all components of x and returns the result in the N\_Vector

 $z: z_i = x_i + b, i = 0, \dots, n-1.$ 

Arguments x (N\_Vector) a NVECTOR object

b (realtype) constant added to all components of  $\boldsymbol{x}$ 

z (N\_Vector) a NVECTOR object containing the result

Return value None

 $F2003\ Name\ FN_VAddConst$ 

N\_VDotProd

Call d = N\_VDotProd(x, y);

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

Arguments x (N\_Vector) a NVECTOR object with y

y (N\_Vector) a NVECTOR object with x

Return value realtype

 $F2003 \; Name \; FN_VDotProd$ 

#### N\_VMaxNorm

Description Returns the maximum norm of the N-Vector x:  $m = \max_i |x_i|$ .

Arguments x (N\_Vector) a NVECTOR object

Return value realtype

F2003 Name FN\_VMaxNorm

#### N\_VWrmsNorm

Call m = N\_VWrmsNorm(x, w)

Description Returns the weighted root-mean-square norm of the N\_Vector x with realtype weight

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

Arguments x (N\_Vector) a NVECTOR object

w (N\_Vector) a NVECTOR object containing weights

Return value realtype

F2003 Name FN\_VWrmsNorm

#### N\_VWrmsNormMask

Description Returns the weighted root mean square norm of the N\_Vector x with realtype weight

vector w built using only the elements of x corresponding to positive elements of the

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

Arguments x (N\_Vector) a NVECTOR object

w (N\_Vector) a NVECTOR object containing weights

id (N\_Vector) mask vector

Return value realtype

F2003 Name FN\_VWrmsNormMask

#### N\_VMin

Call  $m = N_VMin(x);$ 

Description Returns the smallest element of the N\_Vector x:  $m = \min_i x_i$ .

Arguments x (N\_Vector) a NVECTOR object

Return value realtype

F2003 Name FN\_VMin

#### N\_VWL2Norm

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

Description Returns the weighted Euclidean  $\ell_2$  norm of the N\_Vector x with realtype weight vector

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

Arguments x (N\_Vector) a NVECTOR object

w (N\_Vector) a NVECTOR object containing weights

Return value realtype F2003 Name FN\_VWL2Norm

#### N\_VL1Norm

Call m = N\_VL1Norm(x);

Description Returns the  $\ell_1$  norm of the N\_Vector x:  $m = \sum_{i=0}^{n-1} |x_i|$ . Arguments x (N\_Vector) a NVECTOR object to obtain the norm of

Return value realtype F2003 Name FN\_VL1Norm

#### N\_VCompare

Call N\_VCompare(c, x, z);

Description Compares the components of the N\_Vector x to the realtype scalar c and returns an

N-Vector z such that:  $z_i = 1.0$  if  $|x_i| \ge c$  and  $z_i = 0.0$  otherwise.

Arguments c (realtype) constant that each component of x is compared to

x (N\_Vector) a NVECTOR object

z (N\_Vector) a NVECTOR object containing the result

Return value None

F2003 Name FN\_VCompare

#### N\_VInvTest

Call t = N\_VInvTest(x, z);

Description Sets the components of the N\_Vector z to be the inverses of the components of the

N\_Vector x, with prior testing for zero values:  $z_i = 1.0/x_i$ , i = 0, ..., n-1.

Arguments x (N\_Vector) a NVECTOR object

z (N\_Vector) an output NVECTOR object

Return value Returns a booleantype with value SUNTRUE if all components of x are nonzero (success-

ful inversion) and returns SUNFALSE otherwise.

 $F2003 \ Name \ FN_VInvTest$ 

# N\_VConstrMask

Call t = N\_VConstrMask(c, x, m);

Description Performs the following constraint tests:  $x_i > 0$  if  $c_i = 2$ ,  $x_i \geq 0$  if  $c_i = 1$ ,  $x_i \leq 0$  if

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

Arguments c (realtype) scalar constraint value

x (N\_Vector) a NVECTOR object
m (N\_Vector) output mask vector

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

F2003 Name FN\_VConstrMask

#### N\_VMinQuotient

Call minq = N\_VMinQuotient(num, denom);

Description This routine returns the minimum of the quotients obtained by term-wise dividing  $\mathtt{num}_i$ 

by denom<sub>i</sub>. A zero element in denom will be skipped. If no such quotients are found, then the large value BIG\_REAL (defined in the header file sundials\_types.h) is returned.

Arguments num (N\_Vector) a NVECTOR object used as the numerator

denom (N\_Vector) a NVECTOR object used as the denominator

Return value realtype

F2003 Name FN\_VMinQuotient

#### 7.1.2 NVECTOR fused functions

Fused and vector array operations are intended to increase data reuse, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. If a particular NVECTOR implementation defines a fused or vector array operation as NULL, the generic NVECTOR module will automatically call standard vector operations as necessary to complete the desired operation. In all SUNDIALS-provided NVECTOR implementations, all fused and vector array operations are disabled by default. However, these implementations provide additional user-callable functions to enable/disable any or all of the fused and vector array operations. See the following sections for the implementation specific functions to enable/disable operations.

#### N\_VLinearCombination

Call ier = N\_VLinearCombination(nv, c, X, z);

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

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

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

Arguments nv (int) the number of vectors in the linear combination

c (realtype\*) an array of  $n_v$  scalars used to scale the corresponding vector in X

X (N\_Vector\*) an array of  $n_v$  NVECTOR objects to be scaled and combined

z (N\_Vector) a NVECTOR object containing the result

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

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

vector array.

 $F2003\ Name\ FN\_VLinearCombination$ 

F2003 Call real(c\_double) :: c(nv)
type(c\_ptr), target :: X(nv)

type(N\_Vector), pointer :: z

ierr = FN\_VLinearCombination(nv, c, X, z)

#### N\_VScaleAddMulti

Call ier = N\_VScaleAddMulti(nv, c, x, Y, Z);

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

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

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

Arguments nv (int) the number of scalars and vectors in c, Y, and Z

- c (realtype\*) an array of  $n_v$  scalars
- x (N\_Vector) a NVECTOR object to be scaled and added to each vector in Y
- Y (N\_Vector\*) an array of  $n_v$  NVECTOR objects where each vector j will have the vector x scaled by c\_j added to it
- Z (N\_Vector) an output array of  $n_v$  NVECTOR objects

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

 $F2003 \; Name \; FN_VScaleAddMulti$ 

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

#### N\_VDotProdMulti

Call ier = N\_VDotProdMulti(nv, x, Y, d);

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

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

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

Arguments nv (int) the number of vectors in Y

- x (N\_Vector) a NVECTOR object to be used in a dot product with each of the vectors in Y
- Y (N\_Vector\*) an array of  $n_v$  NVECTOR objects to use in a dot product with x
- d (realtype\*) an output array of  $n_v$  dot products

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

F2003 Name FN\_VDotProdMulti

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

#### 7.1.3 NVECTOR vector array functions

#### N\_VLinearSumVectorArray

Call ier = N\_VLinearSumVectorArray(nv, a, X, b, Y, Z);

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

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

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

Arguments nv (int) the number of vectors in the vector arrays

a (realtype) constant to scale each vector in X by

X (N\_Vector\*) an array of  $n_v$  NVECTOR objects

Y (N\_Vector\*) an array of  $n_v$  NVECTOR objects

Z (N\_Vector\*) an output array of  $n_v$  NVECTOR objects

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

F2003 Name FN\_VLinearSumVectorArray

# N\_VScaleVectorArray

Call ier = N\_VScaleVectorArray(nv, c, X, Z);

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

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

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

Arguments nv (int) the number of vectors in the vector arrays

c (realtype) constant to scale each vector in X by

X (N\_Vector\*) an array of  $n_v$  NVECTOR objects

Z (N\_Vector\*) an output array of  $n_v$  NVECTOR objects

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

F2003 Name FN\_VScaleVectorArray

# $N_VConstVectorArray$

Call ier = N\_VConstVectorArray(nv, c, X);

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

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

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

Arguments nv (int) the number of vectors in X

c (realtype) constant to set every element in every vector of X to

X (N\_Vector\*) an array of  $n_v$  NVECTOR objects

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

F2003 Name FN\_VConstVectorArray

#### N\_VWrmsNormVectorArray

Call ier = N\_VWrmsNormVectorArray(nv, X, W, m);

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

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

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

Arguments nv (int) the number of vectors in the vector arrays

X (N\_Vector\*) an array of  $n_v$  NVECTOR objects

W (N\_Vector\*) an array of  $n_v$  NVECTOR objects

m (realtype\*) an output array of  $n_v$  norms

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

F2003 Name FN\_VWrmsNormVectorArray

#### N\_VWrmsNormMaskVectorArray

Call ier = N\_VWrmsNormMaskVectorArray(nv, X, W, id, m);

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

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

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

Arguments nv (int) the number of vectors in the vector arrays

X (N\_Vector\*) an array of  $n_v$  NVECTOR objects

W (N\_Vector\*) an array of  $n_v$  NVECTOR objects

id (N\_Vector) the mask vector

m (realtype\*) an output array of  $n_v$  norms

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

F2003 Name FN\_VWrmsNormMaskVectorArray

#### N\_VScaleAddMultiVectorArray

Call ier = N\_VScaleAddMultiVectorArray(nv, ns, c, X, YY, ZZ);

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

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

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

Arguments nv (int) the number of vectors in the vector arrays

ns (int) the number of scalars in c and vector arrays in YY and ZZ

c (realtype\*) an array of  $n_s$  scalars

X (N\_Vector\*) an array of  $n_v$  NVECTOR objects

YY (N\_Vector\*\*) an array of  $n_s$  NVECTOR arrays

ZZ (N\_Vector\*\*) an output array of  $n_s$  NVECTOR arrays

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

# $\mid$ N\_VLinearCombinationVectorArray

Call ier = N\_VLinearCombinationVectorArray(nv, ns, c, XX, Z);

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

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

where c is an array of  $n_s$  scalars (type realtype\*), XX (type N\_Vector\*\*) is an array of  $n_s$  vector arrays each containing  $n_v$  vectors to be summed into the output vector array of  $n_v$  vectors Z (type N\_Vector\*). If the output vector array Z is one of the vector arrays in XX, then it must be the first vector array in XX.

Arguments nv (int) the number of vectors in the vector arrays

ns (int) the number of scalars in c and vector arrays in YY and ZZ

c (realtype\*) an array of  $n_s$  scalars

XX (N\_Vector\*\*) an array of  $n_s$  NVECTOR arrays

Z (N\_Vector\*) an output array NVECTOR objects

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

#### 7.1.4 NVECTOR local reduction functions

Local reduction operations are intended to reduce parallel communication on distributed memory systems, particularly when NVECTOR objects are combined together within a

NVECTOR\_MPIMANYVECTOR object (see Section 7.16). If a particular NVECTOR implementation defines a local reduction operation as NULL, the NVECTOR\_MPIMANYVECTOR module will automatically call standard vector reduction operations as necessary to complete the desired operation. All SUNDIALS-provided NVECTOR implementations include these local reduction operations, which may be used as templates for user-defined NVECTOR implementations.

#### N\_VDotProdLocal

Call d = N\_VDotProdLocal(x, y);

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

Arguments x (N\_Vector) a NVECTOR object

y (N\_Vector) a NVECTOR object

Return value realtype

 $F2003 \; \mathrm{Name} \; \; \mathtt{FN\_VDotProdLocal}$ 

#### N\_VMaxNormLocal

Call m = N\_VMaxNormLocal(x);

Description This routine computes the MPI task-local portion of the maximum norm of the N\_Vector

x:

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

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

Arguments x (N\_Vector) a NVECTOR object

Return value realtype

F2003 Name FN\_VMaxNormLocal

#### N\_VMinLocal

Call m = N\_VMinLocal(x);

 $Description \quad This \ routine \ computes \ the \ smallest \ element \ of \ the \ MPI \ task-local \ portion \ of \ the \ \texttt{N\_Vector}$ 

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).

Arguments x (N\_Vector) a NVECTOR object

Return value realtype

 $F2003 \ Name \ FN_VMinLocal$ 

#### N\_VL1NormLocal

Call n = N\_VL1NormLocal(x);

Description This routine computes the MPI task-local portion of the  $\ell_1$  norm of the N\_Vector x:

$$n = \sum_{i=0}^{n_{local}-1} |x_i|,$$

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

Arguments x (N\_Vector) a NVECTOR object

Return value realtype

F2003 Name FN\_VL1NormLocal

#### N\_VWSqrSumLocal

Call s = N\_VWSqrSumLocal(x,w);

Description This routine computes the MPI task-local portion of the weighted squared sum of the  $N_{\text{-}}$ Vector x with weight vector w:

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

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

Arguments x (N\_Vector) a NVECTOR object

w (N\_Vector) a NVECTOR object containing weights

Return value realtype

 $F2003 \; Name \; FN_VWSqrSumLocal$ 

# N\_VWSqrSumMaskLocal

Call s = N\_VWSqrSumMaskLocal(x,w,id);

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

$$m = \sum_{i=0}^{n_{local}-1} (x_i w_i H(id_i))^2, \text{ where } H(\alpha) = \begin{cases} 1 & \alpha > 0 \\ 0 & \alpha \le 0 \end{cases}$$

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

Arguments x (N\_Vector) a NVECTOR object

w (N\_Vector) a NVECTOR object containing weights id (N\_Vector) a NVECTOR object used as a mask

Return value realtype

 $F2003\ Name\ FN_VWSqrSumMaskLocal$ 

#### N\_VInvTestLocal

Call t = N\_VInvTestLocal(x, z);

Description Sets the MPI task-local components of the N\_Vector z to be the inverses of the components of the N\_Vector x, with prior testing for zero values:

$$z_i = 1.0/x_i, i = 0, \dots, n_{local} - 1,$$

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

Arguments x (N\_Vector) a NVECTOR object

z (N\_Vector) an output NVECTOR object

Return value Returns a booleantype with the value SUNTRUE if all task-local components of x are nonzero (successful inversion) and with the value SUNFALSE otherwise.

F2003 Name FN\_VInvTestLocal

#### N\_VConstrMaskLocal

Call  $t = N_VConstrMaskLocal(c,x,m);$ 

Description Performs the following constraint tests:

$$x_i > 0$$
 if  $c_i = 2$ ,  
 $x_i \ge 0$  if  $c_i = 1$ ,  
 $x_i \le 0$  if  $c_i = -1$ ,  
 $x_i < 0$  if  $c_i = -2$ , and  
no test if  $c_i = 0$ ,

for all MPI task-local components of the vectors. It sets a mask vector m, with elements equal to 1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking.

Arguments c (realtype) scalar constraint value

x (N\_Vector) a NVECTOR object
m (N\_Vector) output mask vector

Return value Returns a booleantype with the value SUNFALSE if any task-local element failed the constraint test and the value SUNTRUE if all passed.

F2003 Name FN\_VConstrMaskLocal

#### N\_VMinQuotientLocal

Call ming = N\_VMinQuotientLocal(num,denom);

Description This routine returns the minimum of the quotients obtained by term-wise dividing num;

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.

Arguments num (N\_Vector) a NVECTOR object used as the numerator

denom (N\_Vector) a NVECTOR object used as the denominator

Return value realtype

F2003 Name FN\_VMinQuotientLocal

# 7.1.5 NVECTOR exchange operations

The following vector exchange operations are also *optional* and are intended only for use when interfacing with the XBraid library for parallel-in-time integration. In that setting these operations are required but are otherwise unused by SUNDIALS packages and may be set to NULL. For each operation, we give the function signature, a description of the expected behavior, and an example of the function usage.

# N\_VBufSize

Call flag = N\_VBufSize(N\_Vector x, sunindextype \*size);

Description This routine returns the buffer size need to exchange in the data in the vector  $\mathbf{x}$  between

computational nodes.

Arguments x (N\_Vector) a NVECTOR object

size (sunindextype\*) the size of the message buffer

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

F2003 Name FN\_VBufSize

#### N\_VBufPack

Call flag = N\_VBufPack(N\_Vector x, void \*buf);

Description This routine fills the exchange buffer buf with the vector data in x.

buf (sunindextype\*) the exchange buffer to pack

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

F2003 Name FN\_VBufPack

# N\_VBufUnpack

Call flag = N\_VBufUnpack(N\_Vector x, void \*buf);

Description This routine unpacks the data in the exchange buffer buf into the vector x.

Arguments x (N\_Vector) a NVECTOR object

buf (sunindextype\*) the exchange buffer to unpack

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

 $F2003 \; Name \; FN_VBufUnpack$ 

# 7.1.6 NVECTOR utility functions

To aid in the creation of custom NVECTOR modules the generic NVECTOR module provides three utility functions N\_VNewEmpty, N\_VCopyOps and N\_VFreeEmpty. When used in custom NVECTOR constructors and clone routines these functions will ease the introduction of any new optional vector operations to the NVECTOR API by ensuring only required operations need to be set and all operations are copied when cloning a vector.

To aid the use of arrays of NVECTOR objects, the generic NVECTOR module also provides the utility functions N\_VCloneVectorArray, N\_VCloneVectorArrayEmpty, and N\_VDestroyVectorArray.

#### N\_VNewEmpty

Call v = N\_VNewEmpty();

Description The function N\_VNewEmpty allocates a new generic NVECTOR object and initializes its

content pointer and the function pointers in the operations structure to NULL.

Arguments None

Return value This function returns an N\_Vector object. If an error occurs when allocating the object,

then this routine will return NULL.

F2003 Name FN\_VNewEmpty

#### N\_VCopyOps

Call retval = N\_VCopyOps(w, v);

Description The function N\_VCopyOps copies the function pointers in the ops structure of w into the

ops structure of v.

Arguments w (N\_Vector) the vector to copy operations from

v (N\_Vector) the vector to copy operations to

Return value This returns 0 if successful and a non-zero value if either of the inputs are NULL or the

ops structure of either input is NULL.

F2003 Name FN\_VCopyOps

# $N_VFreeEmpty$

Call N\_VFreeEmpty(v);

Description This routine frees the generic N\_Vector object, under the assumption that any implementation-

specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the ops pointer is NULL, and, if it is not, it will

free it as well.

Arguments v (N\_Vector)

Return value None

F2003 Name FN\_VFreeEmpty

#### N\_VCloneEmptyVectorArray

Call vecarray = N\_VCloneEmptyVectorArray(count, w);

Description Creates an array of count variables of type N\_Vector, each of the same type as the exist-

ing N\_Vector w. It achieves this by calling the implementation-specific N\_VCloneEmpty

operation.

Arguments count (int) the size of the vector array

w (N\_Vector) the vector to clone

Return value Returns an array of count N\_Vector objects if successful, or NULL if an error occurred

while cloning.

#### $N_VCloneVectorArray$

Call vecarray = N\_VCloneVectorArray(count, w);

Description Creates an array of count variables of type N\_Vector, each of the same type as the

existing  $N\_Vector$  w. It achieves this by calling the implementation-specific  $N\_VClone$ 

operation.

Arguments count (int) the size of the vector array

w (N\_Vector) the vector to clone

Return value Returns an array of count N\_Vector objects if successful, or NULL if an error occurred

while cloning.

#### N\_VDestroyVectorArray

Call N\_VDestroyVectorArray(count, w);

Description Destroys (frees) an array of variables of type N\_Vector. It depends on the implementation-

specific N\_VDestroy operation.

Arguments vs (N\_Vector\*) the array of vectors to destroy

count (int) the size of the vector array

Return value None

#### N\_VNewVectorArray

Call vecarray = N\_VNewVectorArray(count);

Description Returns an empty N\_Vector array large enough to hold count N\_Vector objects. This

function is primarily meant for users of the Fortran 2003 interface.

Arguments count (int) the size of the vector array

Return value Returns a N\_Vector\* if successful, Returns NULL if an error occurred.

Notes Users of the Fortran 2003 interface to the N\_VManyVector or N\_VMPIManyVector will need

this to create an array to hold the subvectors. Note that this function does restrict the the max number of subvectors usable with the N\_VManyVector and N\_VMPIManyVector to the max size of an int despite the ManyVector implementations accepting a subvector

count larger than this value.

F2003 Name FN\_VNewVectorArray

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

Table 7.1: Vector Identifications associated with vector kernels supplied with SUNDIALS.

# N\_VGetVecAtIndexVectorArray

Call v = N\_VGetVecAtIndexVectorArray(vecs, index);

Description Returns the N-Vector object stored in the vector array at the provided index. This

function is primarily meant for users of the Fortran 2003 interface.

Arguments vecs (N\_Vector\*) the array of vectors to index

index (int) the index of the vector to return

Return value Returns the N\_Vector object stored in the vector array at the provided index. Returns

NULL if an error occurred.

F2003 Name FN\_VGetVecAtIndexVectorArray

# N\_VSetVecAtIndexVectorArray

Call N\_VSetVecAtIndexVectorArray(vecs, index, v);

Description Sets the N\_Vector object stored in the vector array at the provided index. This function

is primarily meant for users of the Fortran 2003 interface.

Arguments vecs (N\_Vector\*) the array of vectors to index

index (int) the index of the vector to return

(N\_Vector) the vector to store at the index

Return value None

F2003 Name FN\_VSetVecAtIndexVectorArray

#### 7.1.7 NVECTOR identifiers

Each NVECTOR implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 7.1.

# 7.1.8 The generic NVECTOR module implementation

The generic N\_Vector type is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the vector, and an *ops* field pointing to a structure with generic vector operations. The type N\_Vector is defined as

```
typedef struct _generic_N_Vector *N_Vector;
struct _generic_N_Vector {
    void *content;
    struct _generic_N_Vector_Ops *ops;
};
The _generic_N_Vector_Ops structure is essentially a list of pointers to the various actual vector
operations, and is defined as
struct _generic_N_Vector_Ops {
  N_Vector_ID (*nvgetvectorid)(N_Vector);
  N_Vector
               (*nvclone)(N_Vector);
  N_{Vector}
               (*nvcloneempty)(N_Vector);
  void
               (*nvdestroy)(N_Vector);
  void
               (*nvspace)(N_Vector, sunindextype *, sunindextype *);
  realtype*
               (*nvgetarraypointer)(N_Vector);
               (*nvgetdevicearraypointer)(N_Vector);
  realtype*
  void
               (*nvsetarraypointer)(realtype *, N_Vector);
  void*
               (*nvgetcommunicator)(N_Vector);
  sunindextype (*nvgetlength)(N_Vector);
               (*nvlinearsum)(realtype, N_Vector, realtype, N_Vector, N_Vector);
  biov
  void
               (*nvconst)(realtype, N_Vector);
  void
               (*nvprod)(N_Vector, N_Vector, N_Vector);
  void
               (*nvdiv)(N_Vector, N_Vector, N_Vector);
  biov
               (*nvscale)(realtype, N_Vector, N_Vector);
               (*nvabs)(N_Vector, N_Vector);
  biov
  biov
               (*nvinv)(N_Vector, N_Vector);
               (*nvaddconst)(N_Vector, realtype, N_Vector);
  void
  realtype
               (*nvdotprod)(N_Vector, N_Vector);
  realtype
               (*nvmaxnorm)(N_Vector);
               (*nvwrmsnorm)(N_Vector, N_Vector);
  realtype
  realtype
               (*nvwrmsnormmask)(N_Vector, N_Vector, N_Vector);
  realtype
               (*nvmin)(N_Vector);
               (*nvwl2norm)(N_Vector, N_Vector);
  realtype
  realtype
               (*nvl1norm)(N_Vector);
  void
               (*nvcompare)(realtype, N_Vector, N_Vector);
  booleantype
               (*nvinvtest)(N_Vector, N_Vector);
               (*nvconstrmask)(N_Vector, N_Vector, N_Vector);
  booleantype
  realtype
               (*nvminquotient)(N_Vector, N_Vector);
               (*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
  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*);
               (*nvwrmsnomrmaskvectorarray)(int, N_Vector*, N_Vector*, N_Vector,
  int
                                             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);
```

```
(*nvmaxnormlocal)(N_Vector);
  realtype
  realtype
               (*nvminlocal)(N_Vector);
  realtype
               (*nvl1normlocal)(N_Vector);
  booleantype
               (*nvinvtestlocal)(N_Vector, N_Vector);
  booleantype
               (*nvconstrmasklocal)(N_Vector, N_Vector, N_Vector);
               (*nvminquotientlocal)(N_Vector, N_Vector);
  realtype
  realtype
               (*nvwsqrsumlocal)(N_Vector, N_Vector);
               (*nvwsqrsummasklocal(N_Vector, N_Vector, N_Vector);
  realtype
               (*nvbufsize)(N_Vector, sunindextype *);
  int
  int
               (*nvbufpack)(N_Vector, void*);
  int
               (*nvbufunpack)(N_Vector, void*);
};
```

The generic NVECTOR module defines and implements the vector operations acting on an N\_Vector. These routines are nothing but wrappers for the vector operations defined by a particular NVECTOR implementation, which are accessed through the *ops* field of the N\_Vector structure. To illustrate this point we show below the implementation of a typical vector operation from the generic NVECTOR module, namely N\_VScale, which performs the scaling of a vector x by a scalar c:

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

Section 7.1.1 defines a complete list of all standard vector operations defined by the generic NVECTOR module. Sections 7.1.2, 7.1.3 and 7.1.4 list *optional* fused, vector array and local reduction operations, respectively.

The Fortran 2003 interface provides a bind(C) derived-type for the \_generic\_N\_Vector and the \_generic\_N\_Vector\_Ops structures. Their definition is given below.

```
type, bind(C), public :: N_Vector
type(C_PTR), public :: content
type(C_PTR), public :: ops
end type N_Vector
type, bind(C), public :: N_Vector_Ops
type(C_FUNPTR), public :: nvgetvectorid
type(C_FUNPTR), public :: nvclone
type(C_FUNPTR), public :: nvcloneempty
type(C_FUNPTR), public :: nvdestroy
type(C_FUNPTR), public :: nvspace
type(C_FUNPTR), public :: nvgetarraypointer
type(C_FUNPTR), public :: nvsetarraypointer
type(C_FUNPTR), public :: nvgetcommunicator
type(C_FUNPTR), public :: nvgetlength
type(C_FUNPTR), public :: nvlinearsum
type(C_FUNPTR), public :: nvconst
type(C_FUNPTR), public :: nvprod
type(C_FUNPTR), public :: nvdiv
type(C_FUNPTR), public :: nvscale
type(C_FUNPTR), public :: nvabs
type(C_FUNPTR), public :: nvinv
type(C_FUNPTR), public :: nvaddconst
type(C_FUNPTR), public :: nvdotprod
type(C_FUNPTR), public :: nvmaxnorm
type(C_FUNPTR), public :: nvwrmsnorm
```

```
type(C_FUNPTR), public :: nvwrmsnormmask
type(C_FUNPTR), public :: nvmin
type(C_FUNPTR), public :: nvwl2norm
type(C_FUNPTR), public :: nvl1norm
type(C_FUNPTR), public :: nvcompare
type(C_FUNPTR), public :: nvinvtest
type(C_FUNPTR), public :: nvconstrmask
type(C_FUNPTR), public :: nvminquotient
type(C_FUNPTR), public :: nvlinearcombination
type(C_FUNPTR), public :: nvscaleaddmulti
type(C_FUNPTR), public :: nvdotprodmulti
type(C_FUNPTR), public :: nvlinearsumvectorarray
type(C_FUNPTR), public :: nvscalevectorarray
type(C_FUNPTR), public :: nvconstvectorarray
type(C_FUNPTR), public :: nvwrmsnormvectorarray
type(C_FUNPTR), public :: nvwrmsnormmaskvectorarray
type(C_FUNPTR), public :: nvscaleaddmultivectorarray
type(C_FUNPTR), public :: nvlinearcombinationvectorarray
type(C_FUNPTR), public :: nvdotprodlocal
type(C_FUNPTR), public :: nvmaxnormlocal
type(C_FUNPTR), public :: nvminlocal
type(C_FUNPTR), public :: nvl1normlocal
type(C_FUNPTR), public :: nvinvtestlocal
type(C_FUNPTR), public :: nvconstrmasklocal
type(C_FUNPTR), public :: nvminquotientlocal
type(C_FUNPTR), public :: nvwsqrsumlocal
type(C_FUNPTR), public :: nvwsgrsummasklocal
type(C_FUNPTR), public :: nvbufsize
type(C_FUNPTR), public :: nvbufpack
type(C_FUNPTR), public :: nvbufunpack
end type N_Vector_Ops
```

# 7.1.9 Implementing a custom NVECTOR

A particular implementation of the NVECTOR module must:

- Specify the *content* field of N\_Vector.
- Define and implement the vector operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one NVECTOR module (each with different N\_Vector internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free an N\_Vector with the new *content* field and with *ops* pointing to the new vector operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined N\_Vector (e.g., a routine to print the content for debugging purposes).
- Optionally, provide accessor macros as needed for that particular implementation to be used to access different parts in the *content* field of the newly defined N\_Vector.

It is recommended that a user-supplied NVECTOR implementation returns the SUNDIALS\_NVEC\_CUSTOM identifier from the  $N_VGetVectorID$  function.

To aid in the creation of custom NVECTOR modules the generic NVECTOR module provides two utility functions N\_VNewEmpty and N\_VCopyOps. When used in custom NVECTOR constructors and clone routines these functions will ease the introduction of any new optional vector operations to the

NVECTOR API by ensuring only required operations need to be set and all operations are copied when cloning a vector.

#### 7.1.9.1 Support for complex-valued vectors

While SUNDIALS itself is written under an assumption of real-valued data, it does provide limited support for complex-valued problems. However, since none of the built-in NVECTOR modules supports complex-valued data, users must provide a custom NVECTOR implementation for this task. Many of the NVECTOR routines described in Sections 7.1.1-7.1.4 above naturally extend to complex-valued vectors; however, some do not. To this end, we provide the following guidance:

- N\_VMin and N\_VMinLocal should return the minimum of all real components of the vector, i.e.,  $m = \min_i \operatorname{real}(x_i)$ .
- N\_VConst (and similarly N\_VConstVectorArray) should set the real components of the vector to the input constant, and set all imaginary components to zero, i.e.,  $z_i = c + 0j$ , i = 0, ..., n 1.
- N\_VAddConst should only update the real components of the vector with the input constant, leaving all imaginary components unchanged.
- N\_VWrmsNorm, N\_VWrmsNormMask, N\_VWSqrSumLocal and N\_VWSqrSumMaskLocal should assume that all entries of the weight vector w and the mask vector id are real-valued.
- N\_VDotProd should mathematically return a complex number for complex-valued vectors; as this is not possible with SUNDIALS' current realtype, this routine should be set to NULL in the custom NVECTOR implementation.
- N\_VCompare, N\_VConstrMask, N\_VMinQuotient, N\_VConstrMaskLocal and N\_VMinQuotientLocal are ill-defined due to the lack of a clear ordering in the complex plane. These routines should be set to NULL in the custom NVECTOR implementation.

While many SUNDIALS solver modules may be utilized on complex-valued data, others cannot. Specifically, although both SUNNONLINSOL\_NEWTON and SUNNONLINSOL\_FIXEDPOINT may be used with any of the IVP solvers (CVODE, CVODES, IDA, IDAS and ARKODE) for complex-valued problems, the Anderson-acceleration feature SUNNONLINSOL\_FIXEDPOINT cannot be used due to its reliance on N\_VDotProd. By this same logic, the Anderson acceleration feature within KINSOL also will not work with complex-valued vectors.

Similarly, although each package's linear solver interface (e.g., CVLS) may be used on complex-valued problems, none of the built-in SUNMATRIX or SUNLINSOL modules work. Hence a complex-valued user should provide a custom SUNLINSOL (and optionally a custom SUNMATRIX) implementation for solving linear systems, and then attach this module as normal to the package's linear solver interface.

Finally, constraint-handling features of each package cannot be used for complex-valued data, due to the issue of ordering in the complex plane discussed above with N\_VCompare, N\_VConstrMask, N\_VMinQuotient, N\_VConstrMaskLocal and N\_VMinQuotientLocal.

We provide a simple example of a complex-valued example problem, including a custom complex-valued Fortran 2003 NVECTOR module, in the files

examples/arkode/F2003\_custom/ark\_analytic\_complex\_f2003.f90, examples/arkode/F2003\_custom/fnvector\_complex\_mod.f90, and examples/arkode/F2003\_custom/test\_fnvector\_complex\_mod.f90.

# 7.2 NVECTOR functions used by IDA

In Table 7.2 below, we list the vector functions used in the NVECTOR module used by the IDA package. The table also shows, for each function, which of the code modules uses the function. The IDA column shows function usage within the main integrator module, while the remaining columns show function

usage within the IDALS linear solvers interface, the IDABBDPRE preconditioner module, and the FIDA module.

At this point, we should emphasize that the IDA user does not need to know anything about the usage of vector functions by the IDA code modules in order to use IDA. The information is presented as an implementation detail for the interested reader.

IDABBDPRE IDALS FIDA N\_VGetVectorID N\_VGetLength 4  $N_VClone$ N\_VCloneEmpty 1  $N_VDestroy$ N\_VSpace 2 N\_VGetArrayPointer 1  $N_VSetArrayPointer$ 1 N\_VLinearSum  $\checkmark$  $N_{-}VConst$  $\checkmark$  $N_{-}VProd$  $N_{-}VDiv$  $N_{-}VScale$  $N_{-}VAbs$ N\_VInv  $N_VAddConst$  $N_{VMaxNorm}$ N\_VWrmsNorm  $N_{VMin}$ N\_VMinQuotient N\_VConstrMask  $N_{V}$ N\_VCompare  $N_{VLinearCombination}$  $N_VScaleAddMulti$  $N_VDotProdMulti$ 3 N\_VLinearSumVectorArray N\_VScaleVectorArray

Table 7.2: List of vector functions usage by IDA code modules

Special cases (numbers match markings in table):

- 1. These routines are only required if an internal difference-quotient routine for constructing dense or band Jacobian matrices is used.
- 2. This routine is optional, and is only used in estimating space requirements for IDA modules for user feedback.
- 3. The optional function N\_VDotProdMulti is only used when Classical Gram-Schmidt is enabled with SPGMR or SPFGMR. The remaining operations from Tables 7.1.2 and 7.1.3 not listed above are unused and a user-supplied NVECTOR module for IDA could omit these operations.

4. This routine is only used when an iterative or matrix iterative SUNLINSOL module is supplied to IDA.

Of the functions listed in Table 7.1.1, N\_VWL2Norm, N\_VL1Norm, N\_VInvTest, and N\_VGetCommunicator are *not* used by IDA. Therefore a user-supplied NVECTOR module for IDA could omit these functions (although some may be needed by SUNNONLINSOL or SUNLINSOL modules).

# 7.3 The NVECTOR\_SERIAL implementation

The serial implementation of the NVECTOR module provided with SUNDIALS, NVECTOR\_SERIAL, defines the *content* field of N\_Vector to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, and a boolean flag *own\_data* which specifies the ownership of *data*.

```
struct _N_VectorContent_Serial {
   sunindextype length;
   booleantype own_data;
   realtype *data;
};
```

The header file to include when using this module is nvector\_serial.h. The installed module library to link to is libsundials\_nvecserial.lib where .lib is typically .so for shared libraries and .a for static libraries.

# 7.3.1 NVECTOR\_SERIAL accessor macros

The following macros are provided to access the content of an NVECTOR\_SERIAL vector. The suffix  $\_S$  in the names denotes the serial version.

#### • NV\_CONTENT\_S

This routine gives access to the contents of the serial vector  ${\tt N\_Vector}.$ 

The assignment  $v\_cont = NV\_CONTENT\_S(v)$  sets  $v\_cont$  to be a pointer to the serial  $N\_Vector$  content structure.

Implementation:

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

• NV\_OWN\_DATA\_S, NV\_DATA\_S, NV\_LENGTH\_S

These macros give individual access to the parts of the content of a serial N\_Vector.

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

The assignment  $v_len = NV_LENGTH_S(v)$  sets  $v_len$  to be the length of v. On the other hand, the call  $NV_LENGTH_S(v) = len_v$  sets the length of v to be  $len_v$ .

Implementation:

```
#define NV_OWN_DATA_S(v) ( NV_CONTENT_S(v)->own_data )
#define NV_DATA_S(v) ( NV_CONTENT_S(v)->data )
#define NV_LENGTH_S(v) ( NV_CONTENT_S(v)->length )
```

#### • NV\_Ith\_S

This macro gives access to the individual components of the data array of an N\_Vector.

The assignment  $r = NV_{i,i}$  sets r to be the value of the i-th component of v. The assignment  $NV_{i,i} = r$  sets the value of the i-th component of v to be r.

Here i ranges from 0 to n-1 for a vector of length n.

Implementation:

#define NV\_Ith\_S(v,i) ( NV\_DATA\_S(v)[i] )

#### 7.3.2 NVECTOR SERIAL functions

The NVECTOR\_SERIAL module defines serial implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3 and 7.1.4. Their names are obtained from those in these tables by appending the suffix \_Serial (e.g. N\_VDestroy\_Serial). All the standard vector operations listed in 7.1.1 with the suffix \_Serial appended are callable via the FORTRAN 2003 interface by prepending an 'F' (e.g. FN\_VDestroy\_Serial).

The module NVECTOR\_SERIAL provides the following additional user-callable routines:

#### N\_VNew\_Serial

Prototype N\_Vector N\_VNew\_Serial(sunindextype vec\_length);

Description This function creates and allocates memory for a serial N\_Vector. Its only argument is the vector length.

F2003 Name This function is callable as FN\_VNew\_Serial when using the Fortran 2003 interface mod-

#### N\_VNewEmpty\_Serial

Prototype N\_Vector N\_VNewEmpty\_Serial(sunindextype vec\_length);

Description This function creates a new serial N\_Vector with an empty (NULL) data array.

F2003 Name This function is callable as FN\_VNewEmpty\_Serial when using the Fortran 2003 interface module.

# N\_VMake\_Serial

Prototype N\_Vector N\_VMake\_Serial(sunindextype vec\_length, realtype \*v\_data);

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

(This function does *not* allocate memory for v\_data itself.)

F2003 Name This function is callable as FN\_VMake\_Serial when using the Fortran 2003 interface module.

# N\_VCloneVectorArray\_Serial

Prototype N\_Vector \*N\_VCloneVectorArray\_Serial(int count, N\_Vector w);

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

F2003 Name This function is callable as FN\_VCloneVectorArray\_Serial when using the Fortran 2003 interface module.

#### N\_VCloneVectorArrayEmpty\_Serial

Prototype N\_Vector \*N\_VCloneVectorArrayEmpty\_Serial(int count, N\_Vector w);

Description This function creates (by cloning) an array of count serial vectors, each with an empty (NULL) data array.

F2003 Name This function is callable as FN\_VCloneVectorArrayEmpty\_Serial when using the Fortran 2003 interface module.

# N\_VDestroyVectorArray\_Serial

Prototype void N\_VDestroyVectorArray\_Serial(N\_Vector \*vs, int count);

Description This function frees memory allocated for the array of count variables of type N\_Vector

created with N\_VCloneVectorArray\_Serial or with

N\_VCloneVectorArrayEmpty\_Serial.

F2003 Name This function is callable as FN\_VDestroyVectorArray\_Serial when using the Fortran

2003 interface module.

#### N\_VPrint\_Serial

Prototype void N\_VPrint\_Serial(N\_Vector v);

Description This function prints the content of a serial vector to stdout.

F2003 Name This function is callable as FN\_VPrint\_Serial when using the Fortran 2003 interface

module.

#### N\_VPrintFile\_Serial

Prototype void N\_VPrintFile\_Serial(N\_Vector v, FILE \*outfile);

Description This function prints the content of a serial vector to outfile.

F2003 Name This function is callable as FN\_VPrintFile\_Serial when using the Fortran 2003 interface

module.

By default all fused and vector array operations are disabled in the NVECTOR\_SERIAL module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with N\_VNew\_Serial, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using N\_VClone. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with N\_VNew\_Serial will have the default settings for the NVECTOR\_SERIAL module.

#### N\_VEnableFusedOps\_Serial

Prototype int N\_VEnableFusedOps\_Serial(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array op-

erations in the serial vector. The return value is 0 for success and -1 if the input vector

or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableFusedOps\_Serial when using the Fortran 2003

interface module.

# N\_VEnableLinearCombination\_Serial

Prototype int N\_VEnableLinearCombination\_Serial(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused

operation in the serial vector. The return value is 0 for success and -1 if the input

vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableLinearCombination\_Serial when using the For-

tran 2003 interface module.

# N\_VEnableScaleAddMulti\_Serial

Prototype int N\_VEnableScaleAddMulti\_Serial(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableScaleAddMulti\_Serial when using the Fortran 2003 interface module.

## N\_VEnableDotProdMulti\_Serial

Prototype int N\_VEnableDotProdMulti\_Serial(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableDotProdMulti\_Serial when using the Fortran 2003 interface module.

#### N\_VEnableLinearSumVectorArray\_Serial

Prototype int N\_VEnableLinearSumVectorArray\_Serial(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableLinearSumVectorArray\_Serial when using the Fortran 2003 interface module.

## N\_VEnableScaleVectorArray\_Serial

Prototype int N\_VEnableScaleVectorArray\_Serial(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableScaleVectorArray\_Serial when using the Fortran 2003 interface module.

# N\_VEnableConstVectorArray\_Serial

Prototype int N\_VEnableConstVectorArray\_Serial(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableConstVectorArray\_Serial when using the Fortran 2003 interface module.

#### N\_VEnableWrmsNormVectorArray\_Serial

Prototype int N\_VEnableWrmsNormVectorArray\_Serial(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableWrmsNormVectorArray\_Serial when using the Fortran 2003 interface module.

# N\_VEnableWrmsNormMaskVectorArray\_Serial

Prototype int N\_VEnableWrmsNormMaskVectorArray\_Serial(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableWrmsNormMaskVectorArray\_Serial when using the Fortran 2003 interface module.

### N\_VEnableScaleAddMultiVectorArray\_Serial

Prototype int N\_VEnableScaleAddMultiVectorArray\_Serial(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

# N\_VEnableLinearCombinationVectorArray\_Serial

Prototype int N\_VEnableLinearCombinationVectorArray\_Serial(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

#### Notes

• When looping over the components of an N\_Vector v, it is more efficient to first obtain the component array via v\_data = NV\_DATA\_S(v) and then access v\_data[i] within the loop than it is to use NV\_Ith\_S(v,i) within the loop.



• N\_VNewEmpty\_Serial, N\_VMake\_Serial, and N\_VCloneVectorArrayEmpty\_Serial set the field  $own\_data = SUNFALSE$ . N\_VDestroy\_Serial and N\_VDestroyVectorArray\_Serial will not attempt to free the pointer data for any N\_Vector with  $own\_data$  set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the data pointer.



• To maximize efficiency, vector operations in the NVECTOR\_SERIAL implementation that have more than one N\_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N\_Vector arguments that were all created with the same internal representations.

# 7.3.3 NVECTOR\_SERIAL Fortran interfaces

The NVECTOR\_SERIAL module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

#### FORTRAN 2003 interface module

The fnvector\_serial\_mod FORTRAN module defines interfaces to all NVECTOR\_SERIAL C functions using the intrinsic iso\_c\_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function N\_VNew\_Serial is interfaced as FN\_VNew\_Serial.

The FORTRAN 2003 NVECTOR\_SERIAL interface module can be accessed with the use statement, i.e. use fnvector\_serial\_mod, and linking to the library libsundials\_fnvectorserial\_mod.lib in addition to the C library. For details on where the library and module file fnvector\_serial\_mod.mod are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the libsundials\_fnvectorserial\_mod library.

#### FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the NVECTOR\_SERIAL module also includes a FORTRAN-callable function FNVINITS(code, NEQ, IER), to initialize this NVECTOR\_SERIAL module. Here code is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); NEQ is the problem size (declared so as to match C type long int); and IER is an error return flag equal 0 for success and -1 for failure.

# 7.4 The NVECTOR\_PARALLEL implementation

The NVECTOR\_PARALLEL implementation of the NVECTOR module provided with SUNDIALS is based on MPI. It defines the *content* field of N\_Vector to be a structure containing the global and local lengths of the vector, a pointer to the beginning of a contiguous local data array, an MPI communicator, and a boolean flag own\_data indicating ownership of the data array data.

```
struct _N_VectorContent_Parallel {
   sunindextype local_length;
   sunindextype global_length;
   booleantype own_data;
   realtype *data;
   MPI_Comm comm;
};
```

The header file to include when using this module is nvector\_parallel.h. The installed module library to link to is libsundials\_nvecparallel.lib where .lib is typically .so for shared libraries and .a for static libraries.

## 7.4.1 NVECTOR\_PARALLEL accessor macros

The following macros are provided to access the content of a NVECTOR\_PARALLEL vector. The suffix \_P in the names denotes the distributed memory parallel version.

#### • NV\_CONTENT\_P

This macro gives access to the contents of the parallel vector  $N_{-}$ Vector.

The assignment v\_cont = NV\_CONTENT\_P(v) sets v\_cont to be a pointer to the N\_Vector content structure of type struct \_N\_VectorContent\_Parallel.

Implementation:

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

• NV\_OWN\_DATA\_P, NV\_DATA\_P, NV\_LOCLENGTH\_P, NV\_GLOBLENGTH\_P

These macros give individual access to the parts of the content of a parallel N\_Vector.

The assignment  $v_{data} = NV_DATA_P(v)$  sets  $v_{data}$  to be a pointer to the first component of the local data for the  $N_Vector v$ . The assignment  $NV_DATA_P(v) = v_{data}$  sets the component array of v to be  $v_{data}$  by storing the pointer  $v_{data}$ .

The assignment v\_llen = NV\_LOCLENGTH\_P(v) sets v\_llen to be the length of the local part of v. The call NV\_LENGTH\_P(v) =  $llen_v$  sets the local length of v to be  $llen_v$ .

The assignment  $v_glen = NV_GLOBLENGTH_P(v)$  sets  $v_glen$  to be the global length of the vector v. The call  $NV_GLOBLENGTH_P(v) = glen_v$  sets the global length of v to be  $glen_v$ .

Implementation:

```
#define NV_OWN_DATA_P(v) ( NV_CONTENT_P(v)->own_data )
#define NV_DATA_P(v) ( NV_CONTENT_P(v)->data )
#define NV_LOCLENGTH_P(v) ( NV_CONTENT_P(v)->local_length )
#define NV_GLOBLENGTH_P(v) ( NV_CONTENT_P(v)->global_length )
```

#### NV\_COMM\_P

This macro provides access to the MPI communicator used by the NVECTOR\_PARALLEL vectors. Implementation:

```
#define NV_COMM_P(v) ( NV_CONTENT_P(v)->comm )
```

#### • NV\_Ith\_P

This macro gives access to the individual components of the local data array of an N-Vector.

The assignment  $r = NV_{i,i}$  sets r to be the value of the i-th component of the local part of v. The assignment  $NV_{i,i} = r$  sets the value of the i-th component of the local part of v to be r.

Here i ranges from 0 to n-1, where n is the local length.

Implementation:

```
#define NV_Ith_P(v,i) ( NV_DATA_P(v)[i] )
```

#### 7.4.2 NVECTOR\_PARALLEL functions

The NVECTOR\_PARALLEL module defines parallel implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4. Their names are obtained from those in these tables by appending the suffix \_Parallel (e.g. N\_VDestroy\_Parallel). The module NVECTOR\_PARALLEL provides the following additional user-callable routines:

#### N\_VNew\_Parallel

```
Prototype N_Vector N_VNew_Parallel(MPI_Comm comm, sunindextype local_length, sunindextype global_length);
```

Description This function creates and allocates memory for a parallel vector.

F2003 Name This function is callable as FN\_VNew\_Parallel when using the Fortran 2003 interface module.

# N\_VNewEmpty\_Parallel

```
Prototype N_Vector N_VNewEmpty_Parallel(MPI_Comm comm, sunindextype local_length, sunindextype global_length);
```

Description This function creates a new parallel N\_Vector with an empty (NULL) data array.

F2003 Name This function is callable as FN\_VNewEmpty\_Parallel when using the Fortran 2003 interface module.

### N\_VMake\_Parallel

Prototype N\_Vector N\_VMake\_Parallel(MPI\_Comm comm, sunindextype local\_length, sunindextype global\_length, realtype \*v\_data);

Description This function creates and allocates memory for a parallel vector with user-provided data array. This function does *not* allocate memory for v\_data itself.

F2003 Name This function is callable as FN\_VMake\_Parallel when using the Fortran 2003 interface module.

# N\_VCloneVectorArray\_Parallel

Prototype N\_Vector \*N\_VCloneVectorArray\_Parallel(int count, N\_Vector w);

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

F2003 Name This function is callable as FN\_VCloneVectorArray\_Parallel when using the Fortran 2003 interface module.

## N\_VCloneVectorArrayEmpty\_Parallel

Prototype N\_Vector \*N\_VCloneVectorArrayEmpty\_Parallel(int count, N\_Vector w);

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

F2003 Name This function is callable as FN\_VCloneVectorArrayEmpty\_Parallel when using the Fortran 2003 interface module.

### N\_VDestroyVectorArray\_Parallel

Prototype void N\_VDestroyVectorArray\_Parallel(N\_Vector \*vs, int count);

Description This function frees memory allocated for the array of count variables of type N\_Vector created with N\_VCloneVectorArray\_Parallel or with N\_VCloneVectorArrayEmpty\_Parallel.

F2003 Name This function is callable as FN\_VDestroyVectorArray\_Parallel when using the Fortran 2003 interface module.

# N\_VGetLocalLength\_Parallel

Prototype sunindextype N\_VGetLocalLength\_Parallel(N\_Vector v);

Description This function returns the local vector length.

F2003 Name This function is callable as FN\_VGetLocalLength\_Parallel when using the Fortran 2003 interface module.

#### N\_VPrint\_Parallel

Prototype void N\_VPrint\_Parallel(N\_Vector v);

Description This function prints the local content of a parallel vector to stdout.

F2003 Name This function is callable as FN\_VPrint\_Parallel when using the Fortran 2003 interface module.

### N\_VPrintFile\_Parallel

Prototype void N\_VPrintFile\_Parallel(N\_Vector v, FILE \*outfile);

Description This function prints the local content of a parallel vector to outfile.

F2003 Name This function is callable as FN\_VPrintFile\_Parallel when using the Fortran 2003 interface module.

By default all fused and vector array operations are disabled in the NVECTOR\_PARALLEL module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with N\_VNew\_Parallel, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using N\_VClone with that vector. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with N\_VNew\_Parallel will have the default settings for the NVECTOR\_PARALLEL module.

#### N\_VEnableFusedOps\_Parallel

Prototype int N\_VEnableFusedOps\_Parallel(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableFusedOps\_Parallel when using the Fortran 2003 interface module.

### N\_VEnableLinearCombination\_Parallel

Prototype int N\_VEnableLinearCombination\_Parallel(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableLinearCombination\_Parallel when using the Fortran 2003 interface module.

# N\_VEnableScaleAddMulti\_Parallel

Prototype int N\_VEnableScaleAddMulti\_Parallel(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableScaleAddMulti\_Parallel when using the Fortran 2003 interface module.

## N\_VEnableDotProdMulti\_Parallel

Prototype int N\_VEnableDotProdMulti\_Parallel(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableDotProdMulti\_Parallel when using the Fortran 2003 interface module.

# N\_VEnableLinearSumVectorArray\_Parallel

Prototype int N\_VEnableLinearSumVectorArray\_Parallel(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableLinearSumVectorArray\_Parallel when using the Fortran 2003 interface module.

### N\_VEnableScaleVectorArray\_Parallel

Prototype int N\_VEnableScaleVectorArray\_Parallel(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableScaleVectorArray\_Parallel when using the Fortran 2003 interface module.

## N\_VEnableConstVectorArray\_Parallel

Prototype int N\_VEnableConstVectorArray\_Parallel(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableConstVectorArray\_Parallel when using the Fortran 2003 interface module.

#### N\_VEnableWrmsNormVectorArray\_Parallel

Prototype int N\_VEnableWrmsNormVectorArray\_Parallel(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableWrmsNormVectorArray\_Parallel when using the Fortran 2003 interface module.

# N\_VEnableWrmsNormMaskVectorArray\_Parallel

Prototype int N\_VEnableWrmsNormMaskVectorArray\_Parallel(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableWrmsNormMaskVectorArray\_Parallel when using the Fortran 2003 interface module.

#### N\_VEnableScaleAddMultiVectorArray\_Parallel

 $Prototype \quad \text{ int $N_V$EnableScaleAddMultiVectorArray\_Parallel($N_V$ector $v$,} \\$ 

booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

```
N_VEnableLinearCombinationVectorArray_Parallel
```

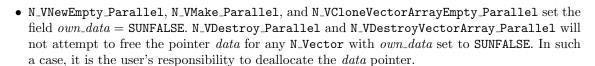
Prototype int N\_VEnableLinearCombinationVectorArray\_Parallel(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the

input vector or its ops structure are NULL.

#### Notes

• When looping over the components of an N\_Vector v, it is more efficient to first obtain the local component array via v\_data = NV\_DATA\_P(v) and then access v\_data[i] within the loop than it is to use NV\_Ith\_P(v,i) within the loop.



• To maximize efficiency, vector operations in the NVECTOR\_PARALLEL implementation that have more than one N\_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N\_Vector arguments that were all created with the same internal representations.

# 7.4.3 NVECTOR\_PARALLEL Fortran interfaces

For solvers that include a FORTRAN 77 interface module, the NVECTOR\_PARALLEL module also includes a FORTRAN-callable function FNVINITP(COMM, code, NLOCAL, NGLOBAL, IER), to initialize this NVECTOR\_PARALLEL module. Here COMM is the MPI communicator, code is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); NLOCAL and NGLOBAL are the local and global vector sizes, respectively (declared so as to match C type long int); and IER is an error return flag equal 0 for success and -1 for failure. NOTE: If the header file sundials\_config.h defines SUNDIALS\_MPI\_COMM\_F2C to be 1 (meaning the MPI implementation used to build SUNDIALS includes the MPI\_Comm\_f2c function), then COMM can be any valid MPI communicator. Otherwise, MPI\_COMM\_WORLD will be used, so just pass an integer value as a placeholder.

# 7.5 The NVECTOR\_OPENMP implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR\_OPENMP, and an implementation using Pthreads, called NVECTOR\_PTHREADS. Testing has shown that vectors should be of length at least 100,000 before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The OpenMP NVECTOR implementation provided with SUNDIALS, NVECTOR\_OPENMP, defines the content field of N\_Vector to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag own\_data which specifies the ownership of data, and the number of threads. Operations on the vector are threaded using OpenMP.

```
struct _N_VectorContent_OpenMP {
   sunindextype length;
   booleantype own_data;
   realtype *data;
   int num_threads;
};
```







The header file to include when using this module is nvector\_openmp.h. The installed module library to link to is libsundials\_nvecopenmp.lib where .lib is typically .so for shared libraries and .a for static libraries. The FORTRAN module file to use when using the FORTRAN 2003 interface to this module is fnvector\_openmp\_mod.mod.

#### 7.5.1 NVECTOR\_OPENMP accessor macros

The following macros are provided to access the content of an NVECTOR\_OPENMP vector. The suffix \_OMP in the names denotes the OpenMP version.

#### NV\_CONTENT\_OMP

This routine gives access to the contents of the OpenMP vector N\_Vector.

The assignment  $v\_cont = NV\_CONTENT\_OMP(v)$  sets  $v\_cont$  to be a pointer to the OpenMP  $N\_Vector$  content structure.

Implementation:

```
#define NV_CONTENT_OMP(v) ( (N_VectorContent_OpenMP)(v->content) )
```

• NV\_OWN\_DATA\_OMP, NV\_DATA\_OMP, NV\_LENGTH\_OMP, NV\_NUM\_THREADS\_OMP

These macros give individual access to the parts of the content of a OpenMP N\_Vector.

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

The assignment  $v_len = NV_length_OMP(v)$  sets  $v_len$  to be the length of v. On the other hand, the call  $NV_length_OMP(v) = len_v$  sets the length of v to be  $len_v$ .

The assignment  $v_num_threads = NV_NUM_THREADS_OMP(v)$  sets  $v_num_threads$  to be the number of threads from v. On the other hand, the call  $NV_NUM_THREADS_OMP(v) = num_threads_v$  sets the number of threads for v to be  $num_threads_v$ .

Implementation:

```
#define NV_OWN_DATA_OMP(v) ( NV_CONTENT_OMP(v)->own_data )
#define NV_DATA_OMP(v) ( NV_CONTENT_OMP(v)->data )
#define NV_LENGTH_OMP(v) ( NV_CONTENT_OMP(v)->length )
#define NV_NUM_THREADS_OMP(v) ( NV_CONTENT_OMP(v)->num_threads )
```

#### • NV\_Ith\_OMP

This macro gives access to the individual components of the data array of an N\_Vector.

The assignment  $r = NV_{in}(v,i)$  sets r to be the value of the i-th component of v. The assignment  $NV_{in}(v,i) = r$  sets the value of the i-th component of v to be r.

Here i ranges from 0 to n-1 for a vector of length n.

Implementation:

```
#define NV_Ith_OMP(v,i) ( NV_DATA_OMP(v)[i] )
```

### 7.5.2 NVECTOR OPENMP functions

The NVECTOR\_OPENMP module defines OpenMP implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4. Their names are obtained from those in these tables by appending the suffix \_OpenMP (e.g. N\_VDestroy\_OpenMP). All the standard vector operations listed in 7.1.1 with the suffix \_OpenMP appended are callable via the FORTRAN 2003 interface by prepending an 'F' (e.g. FN\_VDestroy\_OpenMP).

The module NVECTOR\_OPENMP provides the following additional user-callable routines:

### N\_VNew\_OpenMP

Prototype N\_Vector N\_VNew\_OpenMP(sunindextype vec\_length, int num\_threads)

Description This function creates and allocates memory for a OpenMP N\_Vector. Arguments are the vector length and number of threads.

F2003 Name This function is callable as FN\_VNew\_OpenMP when using the Fortran 2003 interface module

# N\_VNewEmpty\_OpenMP

Prototype N\_Vector N\_VNewEmpty\_OpenMP(sunindextype vec\_length, int num\_threads)

Description This function creates a new OpenMP N\_Vector with an empty (NULL) data array.

F2003 Name This function is callable as FN\_VNewEmpty\_OpenMP when using the Fortran 2003 interface module.

# N\_VMake\_OpenMP

Prototype N\_Vector N\_VMake\_OpenMP(sunindextype vec\_length, realtype \*v\_data, int num\_threads);

Description This function creates and allocates memory for a OpenMP vector with user-provided data array. This function does *not* allocate memory for v\_data itself.

F2003 Name This function is callable as FN\_VMake\_OpenMP when using the Fortran 2003 interface module.

### N\_VCloneVectorArray\_OpenMP

Prototype N\_Vector \*N\_VCloneVectorArray\_OpenMP(int count, N\_Vector w)

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

F2003 Name This function is callable as FN\_VCloneVectorArray\_OpenMP when using the Fortran 2003 interface module.

# N\_VCloneVectorArrayEmpty\_OpenMP

Prototype N\_Vector \*N\_VCloneVectorArrayEmpty\_OpenMP(int count, N\_Vector w)

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

F2003 Name This function is callable as FN\_VCloneVectorArrayEmpty\_OpenMP when using the Fortran 2003 interface module.

# $N_{VDestroyVectorArray_OpenMP}$

Prototype void N\_VDestroyVectorArray\_OpenMP(N\_Vector \*vs, int count)

Description This function frees memory allocated for the array of count variables of type N\_Vector created with N\_VCloneVectorArray\_OpenMP or with N\_VCloneVectorArrayEmpty\_OpenMP.

F2003 Name This function is callable as FN\_VDestroyVectorArray\_OpenMP when using the Fortran 2003 interface module.

### N\_VPrint\_OpenMP

Prototype void N\_VPrint\_OpenMP(N\_Vector v)

Description This function prints the content of an OpenMP vector to stdout.

F2003 Name This function is callable as FN\_VPrint\_OpenMP when using the Fortran 2003 interface

module.

### N\_VPrintFile\_OpenMP

Prototype void N\_VPrintFile\_OpenMP(N\_Vector v, FILE \*outfile)

Description This function prints the content of an OpenMP vector to outfile.

 ${\tt F2003~Name~This~function~is~callable~as~FN\_VPrintFile\_OpenMP~when~using~the~Fortran~2003~interface}$ 

module.

By default all fused and vector array operations are disabled in the NVECTOR\_OPENMP module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with N\_VNew\_OpenMP, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using N\_VClone. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with N\_VNew\_OpenMP will have the default settings for the NVECTOR\_OPENMP module.

#### N\_VEnableFusedOps\_OpenMP

Prototype int N\_VEnableFusedOps\_OpenMP(N\_Vector v, booleantype tf)

 ${\it Description} \quad {\it This function enables ({\tt SUNTRUE}) \ or \ disables \ ({\tt SUNFALSE}) \ all \ fused \ and \ vector \ array \ op-line \ op-li$ 

erations in the OpenMP vector. The return value is 0 for success and -1 if the input

vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableFusedOps\_OpenMP when using the Fortran 2003

interface module.

#### N\_VEnableLinearCombination\_OpenMP

Prototype int N\_VEnableLinearCombination\_OpenMP(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused

operation in the OpenMP vector. The return value is  ${\tt 0}$  for success and  ${\tt -1}$  if the input

vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableLinearCombination\_OpenMP when using the For-

tran 2003 interface module.

# N\_VEnableScaleAddMulti\_OpenMP

Prototype int N\_VEnableScaleAddMulti\_OpenMP(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to

multiple vectors fused operation in the OpenMP vector. The return value is 0 for success

and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableScaleAddMulti\_OpenMP when using the Fortran

2003 interface module.

### N\_VEnableDotProdMulti\_OpenMP

Prototype int N\_VEnableDotProdMulti\_OpenMP(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableDotProdMulti\_OpenMP when using the Fortran 2003 interface module.

# N\_VEnableLinearSumVectorArray\_OpenMP

Prototype int N\_VEnableLinearSumVectorArray\_OpenMP(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableLinearSumVectorArray\_OpenMP when using the Fortran 2003 interface module.

### N\_VEnableScaleVectorArray\_OpenMP

Prototype int N\_VEnableScaleVectorArray\_OpenMP(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableScaleVectorArray\_OpenMP when using the Fortran 2003 interface module.

#### N\_VEnableConstVectorArray\_OpenMP

Prototype int N\_VEnableConstVectorArray\_OpenMP(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableConstVectorArray\_OpenMP when using the Fortran 2003 interface module.

### | N\_VEnableWrmsNormVectorArray\_OpenMP

 $\label{lem:prototype} Prototype \quad \text{int $N_V$EnableWrmsNormVectorArray_OpenMP($N_V$ector $v$, booleantype tf)}$ 

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableWrmsNormVectorArray\_OpenMP when using the Fortran 2003 interface module.

#### N\_VEnableWrmsNormMaskVectorArray\_OpenMP

Prototype int N\_VEnableWrmsNormMaskVectorArray\_OpenMP(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableWrmsNormMaskVectorArray\_OpenMP when using the Fortran 2003 interface module.

### N\_VEnableScaleAddMultiVectorArray\_OpenMP

Prototype int N\_VEnableScaleAddMultiVectorArray\_OpenMP(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

#### N\_VEnableLinearCombinationVectorArray\_OpenMP

Prototype int N\_VEnableLinearCombinationVectorArray\_OpenMP(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

#### Notes

- When looping over the components of an N\_Vector v, it is more efficient to first obtain the component array via v\_data = NV\_DATA\_OMP(v) and then access v\_data[i] within the loop than it is to use NV\_Ith\_OMP(v,i) within the loop.
- N\_VNewEmpty\_OpenMP, N\_VMake\_OpenMP, and N\_VCloneVectorArrayEmpty\_OpenMP set the field  $own\_data = SUNFALSE$ . N\_VDestroy\_OpenMP and N\_VDestroyVectorArray\_OpenMP will not attempt to free the pointer data for any N\_Vector with  $own\_data$  set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the data pointer.
- To maximize efficiency, vector operations in the NVECTOR\_OPENMP implementation that have more than one N\_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N\_Vector arguments that were all created with the same internal representations.

#### 7.5.3 NVECTOR OPENMP Fortran interfaces

The NVECTOR\_OPENMP module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

#### FORTRAN 2003 interface module

The nvector\_openmp\_mod FORTRAN module defines interfaces to most NVECTOR\_OPENMP C functions using the intrinsic iso\_c\_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function N\_VNew\_OpenMP is interfaced as FN\_VNew\_OpenMP.

The FORTRAN 2003 NVECTOR\_OPENMP interface module can be accessed with the use statement, i.e. use fnvector\_openmp\_mod, and linking to the library libsundials\_fnvectoropenmp\_mod.lib in addition to the C library. For details on where the library and module file fnvector\_openmp\_mod.mod are installed see Appendix A.





#### FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the NVECTOR\_OPENMP module also includes a FORTRAN-callable function FNVINITOMP(code, NEQ, NUMTHREADS, IER), to initialize this module. Here code is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); NEQ is the problem size (declared so as to match C type long int); NUMTHREADS is the number of threads; and IER is an error return flag equal 0 for success and -1 for failure.

# 7.6 The NVECTOR\_PTHREADS implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR\_OPENMP, and an implementation using Pthreads, called NVECTOR\_PTHREADS. Testing has shown that vectors should be of length at least 100,000 before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The Pthreads NVECTOR implementation provided with SUNDIALS, denoted NVECTOR\_PTHREADS, defines the *content* field of N\_Vector to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag *own\_data* which specifies the ownership of *data*, and the number of threads. Operations on the vector are threaded using POSIX threads (Pthreads).

```
struct _N_VectorContent_Pthreads {
  sunindextype length;
  booleantype own_data;
  realtype *data;
  int num_threads;
};
```

The header file to include when using this module is nvector\_pthreads.h. The installed module library to link to is libsundials\_nvecpthreads.lib where .lib is typically .so for shared libraries and .a for static libraries.

# 7.6.1 NVECTOR\_PTHREADS accessor macros

The following macros are provided to access the content of an NVECTOR\_PTHREADS vector. The suffix \_PT in the names denotes the Pthreads version.

# • NV\_CONTENT\_PT

This routine gives access to the contents of the Pthreads vector N\_Vector.

The assignment  $v\_cont = NV\_CONTENT\_PT(v)$  sets  $v\_cont$  to be a pointer to the Pthreads  $N\_Vector$  content structure.

Implementation:

```
#define NV_CONTENT_PT(v) ( (N_VectorContent_Pthreads)(v->content) )
```

• NV\_OWN\_DATA\_PT, NV\_DATA\_PT, NV\_LENGTH\_PT, NV\_NUM\_THREADS\_PT

These macros give individual access to the parts of the content of a Pthreads N\_Vector.

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

The assignment  $v_len = NV_LENGTH_PT(v)$  sets  $v_len$  to be the length of v. On the other hand, the call  $NV_LENGTH_PT(v) = len_v$  sets the length of v to be  $len_v$ .

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

Implementation:

```
#define NV_OWN_DATA_PT(v) ( NV_CONTENT_PT(v)->own_data )
#define NV_DATA_PT(v) ( NV_CONTENT_PT(v)->data )
#define NV_LENGTH_PT(v) ( NV_CONTENT_PT(v)->length )
#define NV_NUM_THREADS_PT(v) ( NV_CONTENT_PT(v)->num_threads )
```

#### • NV\_Ith\_PT

This macro gives access to the individual components of the data array of an N\_Vector.

The assignment  $r = NV_Ith_PT(v,i)$  sets r to be the value of the i-th component of v. The assignment  $NV_Ith_PT(v,i) = r$  sets the value of the i-th component of v to be r.

Here i ranges from 0 to n-1 for a vector of length n.

Implementation:

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

# 7.6.2 NVECTOR\_PTHREADS functions

The NVECTOR\_PTHREADS module defines Pthreads implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4. Their names are obtained from those in these tables by appending the suffix \_Pthreads (e.g. N\_VDestroy\_Pthreads). All the standard vector operations listed in 7.1.1 are callable via the FORTRAN 2003 interface by prepending an 'F' (e.g. FN\_VDestroy\_Pthreads). The module NVECTOR\_PTHREADS provides the following additional user-callable routines:

#### N\_VNew\_Pthreads

Prototype N\_Vector N\_VNew\_Pthreads(sunindextype vec\_length, int num\_threads)

Description This function creates and allocates memory for a Pthreads N\_Vector. Arguments are the vector length and number of threads.

F2003 Name This function is callable as FN\_VNew\_Pthreads when using the Fortran 2003 interface module.

#### N\_VNewEmpty\_Pthreads

Prototype N\_Vector N\_VNewEmpty\_Pthreads(sunindextype vec\_length, int num\_threads)

Description This function creates a new Pthreads N\_Vector with an empty (NULL) data array.

F2003 Name This function is callable as FN\_VNewEmpty\_Pthreads when using the Fortran 2003 interface module.

## N\_VMake\_Pthreads

Prototype N\_Vector N\_VMake\_Pthreads(sunindextype vec\_length, realtype \*v\_data, int num\_threads);

Description This function creates and allocates memory for a Pthreads vector with user-provided data array. This function does *not* allocate memory for v\_data itself.

F2003 Name This function is callable as FN\_VMake\_Pthreads when using the Fortran 2003 interface module.

# N\_VCloneVectorArray\_Pthreads

 $\label{lem:prototype} $$\operatorname{N\_Vector} *\operatorname{N\_VCloneVectorArray\_Pthreads(int\ count,\ N\_Vector\ w)}$$ 

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

F2003 Name This function is callable as FN\_VCloneVectorArray\_Pthreads when using the Fortran

2003 interface module.

# N\_VCloneVectorArrayEmpty\_Pthreads

Prototype N\_Vector \*N\_VCloneVectorArrayEmpty\_Pthreads(int count, N\_Vector w)

Description This function creates (by cloning) an array of count Pthreads vectors, each with an

empty (NULL) data array.

F2003 Name This function is callable as FN\_VCloneVectorArrayEmpty\_Pthreads when using the For-

tran 2003 interface module.

# $N_VDestroyVectorArray_Pthreads$

Prototype void N\_VDestroyVectorArray\_Pthreads(N\_Vector \*vs, int count)

Description This function frees memory allocated for the array of count variables of type N\_Vector

created with N\_VCloneVectorArray\_Pthreads or with

N\_VCloneVectorArrayEmpty\_Pthreads.

F2003 Name This function is callable as FN\_VDestroyVectorArray\_Pthreads when using the Fortran

2003 interface module.

### N\_VPrint\_Pthreads

Prototype void N\_VPrint\_Pthreads(N\_Vector v)

Description This function prints the content of a Pthreads vector to stdout.

F2003 Name This function is callable as FN\_VPrint\_Pthreads when using the Fortran 2003 interface

module.

# N\_VPrintFile\_Pthreads

Prototype void N\_VPrintFile\_Pthreads(N\_Vector v, FILE \*outfile)

Description This function prints the content of a Pthreads vector to outfile.

F2003 Name This function is callable as FN\_VPrintFile\_Pthreads when using the Fortran 2003 in-

terface module.

By default all fused and vector array operations are disabled in the NVECTOR\_PTHREADS module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with N\_VNew\_Pthreads, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using N\_VClone. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with N\_VNew\_Pthreads will have the default settings for the NVECTOR\_PTHREADS module.

# N\_VEnableFusedOps\_Pthreads

Prototype int N\_VEnableFusedOps\_Pthreads(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the Pthreads vector. The return value is 0 for success and -1 if the input

erations in the Pthreads vector. The return value is 0 for success and -1 if the input

vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableFusedOps\_Pthreads when using the Fortran 2003 interface module.

#### N\_VEnableLinearCombination\_Pthreads

Prototype int N\_VEnableLinearCombination\_Pthreads(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableLinearCombination\_Pthreads when using the Fortran 2003 interface module.

### N\_VEnableScaleAddMulti\_Pthreads

Prototype int N\_VEnableScaleAddMulti\_Pthreads(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableScaleAddMulti\_Pthreads when using the Fortran 2003 interface module.

#### N\_VEnableDotProdMulti\_Pthreads

Prototype int N\_VEnableDotProdMulti\_Pthreads(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableDotProdMulti\_Pthreads when using the Fortran 2003 interface module.

# N\_VEnableLinearSumVectorArray\_Pthreads

Prototype int N\_VEnableLinearSumVectorArray\_Pthreads(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableLinearSumVectorArray\_Pthreads when using the Fortran 2003 interface module.

#### N\_VEnableScaleVectorArray\_Pthreads

Prototype int N\_VEnableScaleVectorArray\_Pthreads(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableScaleVectorArray\_Pthreads when using the Fortran 2003 interface module.

# N\_VEnableConstVectorArray\_Pthreads

Prototype int N\_VEnableConstVectorArray\_Pthreads(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableConstVectorArray\_Pthreads when using the Fortran 2003 interface module.

# N\_VEnableWrmsNormVectorArray\_Pthreads

Prototype int N\_VEnableWrmsNormVectorArray\_Pthreads(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableWrmsNormVectorArray\_Pthreads when using the Fortran 2003 interface module.

#### N\_VEnableWrmsNormMaskVectorArray\_Pthreads

Prototype int N\_VEnableWrmsNormMaskVectorArray\_Pthreads(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableWrmsNormMaskVectorArray\_Pthreads when using the Fortran 2003 interface module.

#### N\_VEnableScaleAddMultiVectorArray\_Pthreads

Prototype int N\_VEnableScaleAddMultiVectorArray\_Pthreads(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

#### N\_VEnableLinearCombinationVectorArray\_Pthreads

Prototype int N\_VEnableLinearCombinationVectorArray\_Pthreads(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the Pthreads vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

#### Notes

- When looping over the components of an N\_Vector v, it is more efficient to first obtain the component array via v\_data = NV\_DATA\_PT(v) and then access v\_data[i] within the loop than it is to use NV\_Ith\_PT(v,i) within the loop.
- N\_VNewEmpty\_Pthreads, N\_VMake\_Pthreads, and N\_VCloneVectorArrayEmpty\_Pthreads set the field own\_data = SUNFALSE. N\_VDestroy\_Pthreads and N\_VDestroyVectorArray\_Pthreads will not attempt to free the pointer data for any N\_Vector with own\_data set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the data pointer.





• To maximize efficiency, vector operations in the NVECTOR\_PTHREADS implementation that have more than one N\_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N\_Vector arguments that were all created with the same internal representations.

#### 7.6.3 NVECTOR\_PTHREADS Fortran interfaces

The NVECTOR\_PTHREADS module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

#### FORTRAN 2003 interface module

The nvector\_pthreads\_mod FORTRAN module defines interfaces to most NVECTOR\_PTHREADS C functions using the intrinsic iso\_c\_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function N\_VNew\_Pthreads is interfaced as FN\_VNew\_Pthreads.

The FORTRAN 2003 NVECTOR\_PTHREADS interface module can be accessed with the use statement, i.e. use fnvector\_pthreads\_mod, and linking to the library libsundials\_fnvectorpthreads\_mod.lib in addition to the C library. For details on where the library and module file fnvector\_pthreads\_mod.mod are installed see Appendix A.

#### FORTRAN 77 interface functions

For solvers that include a FORTRAN interface module, the NVECTOR\_PTHREADS module also includes a FORTRAN-callable function FNVINITPTS(code, NEQ, NUMTHREADS, IER), to initialize this module. Here code is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); NEQ is the problem size (declared so as to match C type long int); NUMTHREADS is the number of threads; and IER is an error return flag equal 0 for success and -1 for failure.

# 7.7 The NVECTOR\_PARHYP implementation

The NVECTOR\_PARHYP implementation of the NVECTOR module provided with SUNDIALS is a wrapper around *hypre*'s ParVector class. Most of the vector kernels simply call *hypre* vector operations. The implementation defines the *content* field of N\_Vector to be a structure containing the global and local lengths of the vector, a pointer to an object of type HYPRE\_ParVector, an MPI communicator, and a boolean flag *own\_parvector* indicating ownership of the *hypre* parallel vector object *x*.

```
struct _N_VectorContent_ParHyp {
   sunindextype local_length;
   sunindextype global_length;
   booleantype own_parvector;
   MPI_Comm comm;
   HYPRE_ParVector x;
};
```

The header file to include when using this module is nvector\_parhyp.h. The installed module library to link to is libsundials\_nvecparhyp.lib where .lib is typically .so for shared libraries and .a for static libraries.

Unlike native SUNDIALS vector types, NVECTOR\_PARHYP does not provide macros to access its member variables. Note that NVECTOR\_PARHYP requires SUNDIALS to be built with MPI support.

# 7.7.1 NVECTOR\_PARHYP functions

The NVECTOR\_PARHYP module defines implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4, except for N\_VSetArrayPointer and N\_VGetArrayPointer, because accessing raw vector data is handled by low-level *hypre* functions. As such, this vector is not available for use with SUNDIALS Fortran interfaces. When access to raw vector data is needed, one should extract the *hypre* vector first, and then use *hypre* methods to access the data. Usage examples of NVECTOR\_PARHYP are provided in the cvAdvDiff\_non\_ph.c example program for CVODE [35] and the ark\_diurnal\_kry\_ph.c example program for ARKODE [42].

The names of parhyp methods are obtained from those in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4 by appending the suffix Parhyp (e.g. N\_VDestroy\_Parhyp). The module NVECTOR\_PARHYP provides the following additional user-callable routines:

# $N_{VN}$ ewEmpty\_ParHyp

Prototype N\_Vector N\_VNewEmpty\_ParHyp(MPI\_Comm comm, sunindextype local\_length, sunindextype global\_length)

Description This function creates a new parhyp  $N_{-}$ Vector with the pointer to the *hypre* vector set to NULL.

# N\_VMake\_ParHyp

Prototype N\_Vector N\_VMake\_ParHyp(HYPRE\_ParVector x)

Description This function creates an N\_Vector wrapper around an existing hypre parallel vector. It does not allocate memory for x itself.

# N\_VGetVector\_ParHyp

Prototype HYPRE\_ParVector N\_VGetVector\_ParHyp(N\_Vector v)

Description This function returns the underlying hypre vector.

# N\_VCloneVectorArray\_ParHyp

Prototype N\_Vector \*N\_VCloneVectorArray\_ParHyp(int count, N\_Vector w)

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

## N\_VCloneVectorArrayEmpty\_ParHyp

Prototype N\_Vector \*N\_VCloneVectorArrayEmpty\_ParHyp(int count, N\_Vector w)

Description This function creates (by cloning) an array of count parallel vectors, each with an empty

(NULL) data array.

# N\_VDestroyVectorArray\_ParHyp

Prototype void N\_VDestroyVectorArray\_ParHyp(N\_Vector \*vs, int count)

Description This function frees memory allocated for the array of count variables of type N\_Vector created with N\_VCloneVectorArray\_ParHyp or with N\_VCloneVectorArrayEmpty\_ParHyp.

#### N\_VPrint\_ParHyp

Prototype void N\_VPrint\_ParHyp(N\_Vector v)

Description This function prints the local content of a parhyp vector to stdout.

# N\_VPrintFile\_ParHyp

Prototype void N\_VPrintFile\_ParHyp(N\_Vector v, FILE \*outfile)

Description This function prints the local content of a parhyp vector to outfile.

By default all fused and vector array operations are disabled in the NVECTOR\_PARHYP module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with N\_VMake\_ParHyp, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using N\_VClone. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with N\_VMake\_ParHyp will have the default settings for the NVECTOR\_PARHYP module.

### N\_VEnableFusedOps\_ParHyp

Prototype int N\_VEnableFusedOps\_ParHyp(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

# N\_VEnableLinearCombination\_ParHyp

Prototype int N\_VEnableLinearCombination\_ParHyp(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

## N\_VEnableScaleAddMulti\_ParHyp

Prototype int N\_VEnableScaleAddMulti\_ParHyp(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

#### N\_VEnableDotProdMulti\_ParHyp

Prototype int N\_VEnableDotProdMulti\_ParHyp(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

# N\_VEnableLinearSumVectorArray\_ParHyp

Prototype int N\_VEnableLinearSumVectorArray\_ParHyp(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

#### N\_VEnableScaleVectorArray\_ParHyp

Prototype int N\_VEnableScaleVectorArray\_ParHyp(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector

arrays in the parhyp vector. The return value is 0 for success and -1 if the input vector

or its  $\ensuremath{\mathsf{ops}}$  structure are NULL.

### N\_VEnableConstVectorArray\_ParHyp

Prototype int N\_VEnableConstVectorArray\_ParHyp(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the parhyp vector. The return value is 0 for success and -1 if the input vector

or its ops structure are NULL.

# N\_VEnableWrmsNormVectorArray\_ParHyp

Prototype int N\_VEnableWrmsNormVectorArray\_ParHyp(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for

vector arrays in the parhyp vector. The return value is  ${\tt 0}$  for success and  ${\tt -1}$  if the input

vector or its ops structure are NULL.

# ${\tt N\_VEnableWrmsNormMaskVectorArray\_ParHyp}$

Prototype int N\_VEnableWrmsNormMaskVectorArray\_ParHyp(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm op-

eration for vector arrays in the parhyp vector. The return value is 0 for success and -1

if the input vector or its ops structure are NULL.

### N\_VEnableScaleAddMultiVectorArray\_ParHyp

Prototype int N\_VEnableScaleAddMultiVectorArray\_ParHyp(N\_Vector v,

booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array

to multiple vector arrays operation in the parhyp vector. The return value is 0 for success

and -1 if the input vector or its ops structure are NULL.

# ${\tt N\_VEnableLinearCombinationVectorArray\_ParHyp}$

Prototype int N\_VEnableLinearCombinationVectorArray\_ParHyp(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation

for vector arrays in the parhyp vector. The return value is 0 for success and -1 if the

input vector or its ops structure are NULL.

#### Notes

- When there is a need to access components of an N\_Vector\_ParHyp, v, it is recommended to extract the hypre vector via x\_vec = N\_VGetVector\_ParHyp(v) and then access components using appropriate hypre functions.
- N\_VNewEmpty\_ParHyp, N\_VMake\_ParHyp, and N\_VCloneVectorArrayEmpty\_ParHyp set the field own\_parvector to SUNFALSE. N\_VDestroy\_ParHyp and N\_VDestroyVectorArray\_ParHyp will not attempt to delete an underlying hypre vector for any N\_Vector with own\_parvector set to SUNFALSE. In such a case, it is the user's responsibility to delete the underlying vector.





• To maximize efficiency, vector operations in the NVECTOR\_PARHYP implementation that have more than one N\_Vector argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with N\_Vector arguments that were all created with the same internal representations.

# 7.8 The NVECTOR\_PETSC implementation

The NVECTOR\_PETSC module is an NVECTOR wrapper around the PETSc vector. It defines the *content* field of a N\_Vector to be a structure containing the global and local lengths of the vector, a pointer to the PETSc vector, an MPI communicator, and a boolean flag *own\_data* indicating ownership of the wrapped PETSc vector.

```
struct _N_VectorContent_Petsc {
   sunindextype local_length;
   sunindextype global_length;
   booleantype own_data;
   Vec *pvec;
   MPI_Comm comm;
};
```

The header file to include when using this module is nvector\_petsc.h. The installed module library to link to is libsundials\_nvecpetsc.lib where .lib is typically .so for shared libraries and .a for static libraries.

Unlike native SUNDIALS vector types, NVECTOR\_PETSC does not provide macros to access its member variables. Note that NVECTOR\_PETSC requires SUNDIALS to be built with MPI support.

# 7.8.1 NVECTOR\_PETSC functions

The NVECTOR\_PETSC module defines implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4, except for N\_VGetArrayPointer and N\_VSetArrayPointer. As such, this vector cannot be used with SUNDIALS Fortran interfaces. When access to raw vector data is needed, it is recommended to extract the PETSc vector first, and then use PETSc methods to access the data. Usage examples of NVECTOR\_PETSC are provided in example programs for IDA [34].

The names of vector operations are obtained from those in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4 by appending the suffix \_Petsc (e.g. N\_VDestroy\_Petsc). The module NVECTOR\_PETSC provides the following additional user-callable routines:

# $N_{-}VNewEmpty_Petsc$

Prototype N\_Vector N\_VNewEmpty\_Petsc(MPI\_Comm comm, sunindextype local\_length, sunindextype global\_length)

Description This function creates a new NVECTOR wrapper with the pointer to the wrapped PETSc vector set to (NULL). It is used by the N\_VMake\_Petsc and N\_VClone\_Petsc implementations.

### N\_VMake\_Petsc

Prototype N\_Vector N\_VMake\_Petsc(Vec \*pvec)

Description This function creates and allocates memory for an NVECTOR\_PETSC wrapper around a user-provided PETSc vector. It does *not* allocate memory for the vector **pvec** itself.

#### N\_VGetVector\_Petsc

Prototype Vec \*N\_VGetVector\_Petsc(N\_Vector v)

Description This function returns a pointer to the underlying PETSc vector.

# N\_VCloneVectorArray\_Petsc

Prototype N\_Vector \*N\_VCloneVectorArray\_Petsc(int count, N\_Vector w)

Description This function creates (by cloning) an array of count NVECTOR\_PETSC vectors.

# N\_VCloneVectorArrayEmpty\_Petsc

Prototype N\_Vector \*N\_VCloneVectorArrayEmpty\_Petsc(int count, N\_Vector w)

Description This function creates (by cloning) an array of count NVECTOR\_PETSC vectors, each with

pointers to Petsc vectors set to (NULL).

### N\_VDestroyVectorArray\_Petsc

Prototype void N\_VDestroyVectorArray\_Petsc(N\_Vector \*vs, int count)

Description This function frees memory allocated for the array of count variables of type N\_Vector

created with N\_VCloneVectorArray\_Petsc or with N\_VCloneVectorArrayEmpty\_Petsc.

#### N\_VPrint\_Petsc

Prototype void N\_VPrint\_Petsc(N\_Vector v)

Description This function prints the global content of a wrapped PETSc vector to stdout.

#### N\_VPrintFile\_Petsc

Prototype void N\_VPrintFile\_Petsc(N\_Vector v, const char fname[])

Description This function prints the global content of a wrapped PETSc vector to fname.

By default all fused and vector array operations are disabled in the NVECTOR\_PETSC module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with N\_VMake\_Petsc, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using N\_VClone. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with N\_VMake\_Petsc will have the default settings for the NVECTOR\_PETSC module.

#### N\_VEnableFusedOps\_Petsc

Prototype int N\_VEnableFusedOps\_Petsc(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array oper-

ations in the PETSc vector. The return value is 0 for success and -1 if the input vector

or its ops structure are NULL.

## N\_VEnableLinearCombination\_Petsc

Prototype int N\_VEnableLinearCombination\_Petsc(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused

operation in the PETSc vector. The return value is 0 for success and -1 if the input

vector or its ops structure are NULL.

# N\_VEnableScaleAddMulti\_Petsc

Prototype int N\_VEnableScaleAddMulti\_Petsc(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

#### N\_VEnableDotProdMulti\_Petsc

Prototype int N\_VEnableDotProdMulti\_Petsc(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

### N\_VEnableLinearSumVectorArray\_Petsc

Prototype int N\_VEnableLinearSumVectorArray\_Petsc(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

#### N\_VEnableScaleVectorArray\_Petsc

Prototype int N\_VEnableScaleVectorArray\_Petsc(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

### N\_VEnableConstVectorArray\_Petsc

Prototype int N\_VEnableConstVectorArray\_Petsc(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

# N\_VEnableWrmsNormVectorArray\_Petsc

Prototype int N\_VEnableWrmsNormVectorArray\_Petsc(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

#### N\_VEnableWrmsNormMaskVectorArray\_Petsc

Prototype int N\_VEnableWrmsNormMaskVectorArray\_Petsc(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

# N\_VEnableScaleAddMultiVectorArray\_Petsc

Prototype int N\_VEnableScaleAddMultiVectorArray\_Petsc(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the PETSc vector. The return value is 0 for success

and -1 if the input vector or its ops structure are NULL.

# $\verb|N_VEnableLinearCombinationVectorArray_Petsc|$

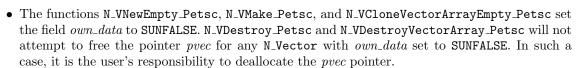
Prototype int N\_VEnableLinearCombinationVectorArray\_Petsc(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation

for vector arrays in the PETSc vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

#### Notes

• When there is a need to access components of an N\_Vector\_Petsc, v, it is recommeded to extract the PETSc vector via x\_vec = N\_VGetVector\_Petsc(v) and then access components using appropriate PETSc functions.



• To maximize efficiency, vector operations in the NVECTOR\_PETSC implementation that have more than one N\_Vector argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with N\_Vector arguments that were all created with the same internal representations.

# 7.9 The NVECTOR\_CUDA implementation

The NVECTOR\_CUDA module is an NVECTOR implementation in the CUDA language. The module allows for SUNDIALS vector kernels to run on NVIDIA GPU devices. It is intended for users who are already familiar with CUDA and GPU programming. Building this vector module requires a CUDA compiler and, by extension, a C++ compiler. The vector content layout is as follows:

```
struct _N_VectorContent_Cuda
  sunindextype
                     length;
  booleantype
                     own_exec;
  booleantype
                     own_helper;
  SUNMemory
                     host_data;
  SUNMemory
                     device_data;
  SUNCudaExecPolicy* stream_exec_policy;
  SUNCudaExecPolicy* reduce_exec_policy;
  SUNMemoryHelper
                     mem_helper;
                     priv; /* 'private' data */
  void*
};
typedef struct _N_VectorContent_Cuda *N_VectorContent_Cuda;
```

The content members are the vector length (size), ownership flags for the \*\_exec\_policy fields and the mem\_helper field, SUNMemory objects for the vector data on the host and the device, pointers to





SUNCudaExecPolicy implementations that control how the CUDA kernels are launched for streaming and reduction vector kernels, a SUNMemoryHelper object, and a private data structure which holds additional members that should not be accessed directly.

When instantiated with N\_VNew\_Cuda, the underlying data will be allocated memory on both the host and the device. Alternatively, a user can provide host and device data arrays by using the N\_VMake\_Cuda constructor. To use CUDA managed memory, the constructors N\_VNewManaged\_Cuda and

N\_VMakeManaged\_Cuda are provided. Details on each of these constructors are provided below.

To use the NVECTOR\_CUDA module, the header file to include is nvector\_cuda.h, and the library to link to is libsundials\_nveccuda.lib. The extension .lib is typically .so for shared libraries and .a for static libraries.

#### 7.9.1 NVECTOR\_CUDA functions

Unlike other native SUNDIALS vector types, NVECTOR\_CUDA does not provide macros to access its member variables. Instead, user should use the accessor functions:

### N\_VGetHostArrayPointer\_Cuda

Prototype realtype \*N\_VGetHostArrayPointer\_Cuda(N\_Vector v)

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

#### N\_VGetDeviceArrayPointer\_Cuda

Prototype realtype \*N\_VGetDeviceArrayPointer\_Cuda(N\_Vector v)

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

#### N\_VSetHostArrayPointer\_Cuda

Prototype realtype \*N\_VSetHostArrayPointer\_Cuda(N\_Vector v)

Description This function sets the pointer to the vector data on the host. The existing pointer will

not be freed first.

# N\_VSetDeviceArrayPointer\_Cuda

Prototype realtype \*N\_VSetDeviceArrayPointer\_Cuda(N\_Vector v)

Description This function sets pointer to the vector data on the device. The existing pointer will

not be freed first.

# N\_VIsManagedMemory\_Cuda

Prototype booleantype \*N\_VIsManagedMemory\_Cuda(N\_Vector v)

Description This function returns a boolean flag indicating if the vector data is allocated in managed

memory or not.

The NVECTOR\_CUDA module defines implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3 and 7.1.4, except for N\_VSetArrayPointer and N\_VGetArrayPointer unless managed memory is used. As such, this vector can only be used with the SUNDIALS Fortran interfaces, and the SUNDIALS direct solvers and preconditioners when using managed memory. The NVECTOR\_CUDA module provides separate functions to access data on the host and on the device for the unmanaged memory use case. It also provides methods for copying from the host to the device and vice versa. Usage examples of NVECTOR\_CUDA are provided in some example programs for CVODE [35].

The names of vector operations are obtained from those in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4 by appending the suffix \_Cuda (e.g. N\_VDestroy\_Cuda). The module NVECTOR\_CUDA provides the following functions:

# N\_VNew\_Cuda

Prototype N\_Vector N\_VNew\_Cuda(sunindextype length)

Description This function creates and allocates memory for a CUDA N\_Vector. The vector data array

is allocated on both the host and device.

#### N\_VNewManaged\_Cuda

Prototype N\_Vector N\_VNewManaged\_Cuda(sunindextype length)

Description This function creates and allocates memory for a CUDA N\_Vector. The vector data array

is allocated in managed memory.

## $N_VNewWithMemHelp_Cuda$

Prototype N\_Vector N\_VNewWithMemHelp\_Cuda(sunindextype length, booleantype use\_managed\_mem,

SUNMemoryHelper helper);

Description This function creates an NVECTOR\_CUDA which will use the SUNMemoryHelper object

to allocate memory. If  ${\tt use\_managed\_memory}$  is 0, then unmanaged memory is used,

otherwise managed memory is used.

#### N\_VNewEmpty\_Cuda

Prototype N\_Vector N\_VNewEmpty\_Cuda()

Description This function creates a new NVECTOR wrapper with the pointer to the wrapped CUDA

vector set to NULL. It is used by the N\_VNew\_Cuda, N\_VMake\_Cuda, and N\_VClone\_Cuda

implementations.

### N\_VMake\_Cuda

Prototype N\_Vector N\_VMake\_Cuda(sunindextype length, realtype \*h\_data, realtype \*dev\_data)

Description This function creates an NVECTOR\_CUDA with user-supplied vector data arrays h\_vdata

and d\_vdata. This function does not allocate memory for data itself.

# $N_VMakeManaged_Cuda$

Prototype N\_Vector N\_VMakeManaged\_Cuda(sunindextype length, realtype \*vdata)

Description This function creates an NVECTOR\_CUDA with a user-supplied managed memory data

array. This function does not allocate memory for data itself.

#### N\_VMakeWithManagedAllocator\_Cuda

Prototype N\_Vector N\_VMakeWithManagedAllocator\_Cuda(sunindextype length, void\* (\*allocfn)(size\_t

size), void (\*freefn)(void\* ptr));

Description This function creates an NVECTOR\_CUDA with a user-supplied memory allocator. It

requires the user to provide a corresponding free function as well. The memory allocated

by the allocator function must behave like CUDA managed memory.

 $This function is deprecated and will be removed in the next major release. \ Use {\tt N\_VNewWithMemHelp\_Cuda} is the second of the control of$ 

instead.

The module NVECTOR\_CUDA also provides the following user-callable routines:



#### N\_VSetKernelExecPolicy\_Cuda

Prototype void N\_VSetKernelExecPolicy\_Cuda(N\_Vector v, SUNCudaExecPolicy\* stream\_exec\_policy, SUNCudaExecPolicy\* reduce\_exec\_policy);

Description

This function sets the execution policies which control the kernel parameters utilized when launching the streaming and reduction CUDA kernels. By default the vector is setup to use the SUNCudaThreadDirectExecPolicy and SUNCudaBlockReduceExecPolicy. Any custom execution policy for reductions must ensure that the grid dimensions (number of thread blocks) is a multiple of the CUDA warp size (32). See section 7.9.2 below for more information about the SUNCudaExecPolicy class.

Note: All vectors used in a single instance of a SUNDIALS solver must use the same execution policy. It is **strongly recommended** that this function is called immediately after constructing the vector, and any subsequent vector be created by cloning to ensure consistent execution policies across vectors.

#### N\_VSetCudaStream\_Cuda

Prototype void N\_VSetCudaStream\_Cuda(N\_Vector v, cudaStream\_t \*stream)

Description This function sets the CUDA stream that all vector kernels will be launched on. By default an NVECTOR\_CUDA uses the default CUDA stream.

Note: All vectors used in a single instance of a SUNDIALS solver must use the same CUDA stream. It is **strongly recommended** that this function is called immediately after constructing the vector, and any subsequent vector be created by cloning to ensure consistent execution policies across vectors.

This function will be removed in the next major release, user should utilize the N\_VSetKernelExecution instead.

#### N\_VCopyToDevice\_Cuda

Prototype void N\_VCopyToDevice\_Cuda(N\_Vector v)

Description This function copies host vector data to the device.

# N\_VCopyFromDevice\_Cuda

Prototype void N\_VCopyFromDevice\_Cuda(N\_Vector v)

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

### N\_VPrint\_Cuda

Prototype void N\_VPrint\_Cuda(N\_Vector v)

Description This function prints the content of a CUDA vector to stdout.

#### N\_VPrintFile\_Cuda

Prototype void N\_VPrintFile\_Cuda(N\_Vector v, FILE \*outfile)

Description This function prints the content of a CUDA vector to outfile.

By default all fused and vector array operations are disabled in the NVECTOR\_CUDA module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with N\_VNew\_Cuda, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using N\_VClone. This guarantees

the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with  $N_{VNew\_Cuda}$  will have the default settings for the  $NVECTOR\_CUDA$  module.

# N\_VEnableFusedOps\_Cuda

Prototype int N\_VEnableFusedOps\_Cuda(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N\_VEnableLinearCombination\_Cuda

Prototype int N\_VEnableLinearCombination\_Cuda(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the CUDA vector. The return value is 0 for success and -1 if the input

vector or its ops structure are NULL.

#### N\_VEnableScaleAddMulti\_Cuda

Prototype int N\_VEnableScaleAddMulti\_Cuda(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to

multiple vectors fused operation in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N\_VEnableDotProdMulti\_Cuda

Prototype int N\_VEnableDotProdMulti\_Cuda(N\_Vector v, booleantype tf)

 $\label{eq:control} \textbf{Description} \quad \textbf{This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused \\$ 

operation in the CUDA vector. The return value is 0 for success and -1 if the input

vector or its ops structure are NULL.

#### N\_VEnableLinearSumVectorArray\_Cuda

Prototype int N\_VEnableLinearSumVectorArray\_Cuda(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for

vector arrays in the CUDA vector. The return value is 0 for success and -1 if the input

vector or its ops structure are NULL.

### N\_VEnableScaleVectorArray\_Cuda

Prototype int N\_VEnableScaleVectorArray\_Cuda(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector

arrays in the CUDA vector. The return value is  ${\tt 0}$  for success and  ${\tt -1}$  if the input vector

or its ops structure are NULL.

#### N\_VEnableConstVectorArray\_Cuda

Prototype int N\_VEnableConstVectorArray\_Cuda(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector

arrays in the CUDA vector. The return value is 0 for success and -1 if the input vector

or its ops structure are NULL.

### N\_VEnableWrmsNormVectorArray\_Cuda

Prototype int N\_VEnableWrmsNormVectorArray\_Cuda(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the CUDA vector. The return value is 0 for success and -1 if the input

vector or its ops structure are NULL.

### N\_VEnableWrmsNormMaskVectorArray\_Cuda

Prototype int N\_VEnableWrmsNormMaskVectorArray\_Cuda(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm op-

eration for vector arrays in the CUDA vector. The return value is 0 for success and -1 if

the input vector or its ops structure are NULL.

### N\_VEnableScaleAddMultiVectorArray\_Cuda

Prototype int N\_VEnableScaleAddMultiVectorArray\_Cuda(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array

to multiple vector arrays operation in the CUDA vector. The return value is  $\boldsymbol{0}$  for success

and -1 if the input vector or its ops structure are NULL.

### ${\tt N\_VEnableLinearCombinationVectorArray\_Cuda}$

Prototype int N\_VEnableLinearCombinationVectorArray\_Cuda(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation

for vector arrays in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

# Notes

- When there is a need to access components of an N\_Vector\_Cuda, v, it is recommeded to use functions N\_VGetDeviceArrayPointer\_Cuda or N\_VGetHostArrayPointer\_Cuda. However, when using managed memory, the function N\_VGetArrayPointer may also be used.
- Performance is better if the SUNMemoryHelper provided supports SUNMEMTYPE\_PINNED; the default SUNMemoryHelper does provide this support. In the case that it does, then the buffers used for reductions will be allocated as pinned memory.
- To maximize efficiency, vector operations in the NVECTOR\_CUDA implementation that have more than one N\_Vector argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with N\_Vector arguments that were all created with the same internal representations.

### 7.9.2 The SUNCudaExecPolicy Class

In order to provide maximum flexibility to users, the CUDA kernel execution parameters used by kernels within SUNDIALS are defined by objects of the sundials::CudaExecPolicy abstract class type (this class can be accessed in the global namespace as SUNCudaExecPolicy). Thus, users may provide custom execution policies that fit the needs of their problem. The sundials::CudaExecPolicy is defined in the header file sundials\_cuda\_policies.hpp, and is as follows:

```
class CudaExecPolicy
{
public:
```



```
virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const = 0;
virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const = 0;
virtual cudaStream_t stream() const = 0;
virtual CudaExecPolicy* clone() const = 0;
virtual ~CudaExecPolicy() {}
};
```

To define a custom execution policy, a user simply needs to create a class that inherits from the abstract class and implements the methods. The SUNDIALS provided sundials::CudaThreadDirectExecPolicy (aka in the global namespace as SUNCudaThreadDirectExecPolicy) class is a good example of a what a custom execution policy may look like:

```
class CudaThreadDirectExecPolicy : public CudaExecPolicy
{
public:
  CudaThreadDirectExecPolicy(const size_t blockDim, const cudaStream_t stream = 0)
    : blockDim_(blockDim), stream_(stream)
  {}
  CudaThreadDirectExecPolicy(const CudaThreadDirectExecPolicy& ex)
    : blockDim_(ex.blockDim_), stream_(ex.stream_)
  {}
  virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const
   return (numWorkUnits + blockSize() - 1) / blockSize();
  virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const
   return blockDim_;
  virtual cudaStream_t stream() const
   return stream_;
  }
  virtual CudaExecPolicy* clone() const
   return static_cast<CudaExecPolicy*>(new CudaThreadDirectExecPolicy(*this));
  }
private:
  const cudaStream_t stream_;
  const size_t blockDim_;
};
```

In total, SUNDIALS provides 3 execution policies:

1. SUNCudaThreadDirectExecPolicy(const size\_t blockDim, const cudaStream\_t stream = 0) maps each CUDA thread to a work unit. The number of threads per block (blockDim) can be set to anything. The grid size will be calculated so that there are enough threads for one thread per element. If a CUDA stream is provided, it will be used to execute the kernel.

- 2. SUNCudaGridStrideExecPolicy(const size\_t blockDim, const size\_t gridDim, const cudaStream\_t stream = 0) is for kernels that use grid stride loops. The number of threads per block (block-Dim) can be set to anything. The number of blocks (gridDim) can be set to anything. If a CUDA stream is provided, it will be used to execute the kernel.
- 3. SUNCudaBlockReduceExecPolicy(const size\_t blockDim, const size\_t gridDim, const cudaStream\_t stream = 0) is for kernels performing a reduction across individual thread blocks. The number of threads per block (blockDim) can be set to any valid multiple of the CUDA warp size. The grid size (gridDim) can be set to any value greater than 0. If it is set to 0, then the grid size will be chosen so that there is enough threads for one thread per work unit. If a CUDA stream is provided, it will be used to execute the kernel.

For example, a policy that uses 128 threads per block and a user provided stream can be created like so:

```
cudaStream_t stream;
cudaStreamCreate(&stream);
SUNCudaThreadDirectExecPolicy thread_direct(128, stream);
```

These default policy objects can be reused for multiple SUNDIALS data structures since they do not hold any modifiable state information.

# 7.10 The NVECTOR\_HIP implementation

The NVECTOR\_HIP module is an NVECTOR implementation using the AMD ROCm HIP library. The module allows for SUNDIALS vector kernels to run on AMD or NVIDIA GPU devices. It is intended for users who are already familiar with HIP and GPU programming. Building this vector module requires the HIP-clang compiler. The vector content layout is as follows:

```
struct _N_VectorContent_Hip
  sunindextype
                     length;
  booleantype
                      own_exec;
  booleantype
                      own_helper;
  SUNMemory
                     host_data;
  SUNMemory
                     device_data;
  SUNHipExecPolicy*
                     stream_exec_policy;
  SUNHipExecPolicy*
                     reduce_exec_policy;
  SUNMemoryHelper
                     mem_helper;
  void*
                     priv; /* 'private' data */
};
```

typedef struct \_N\_VectorContent\_Hip \*N\_VectorContent\_Hip;

The content members are the vector length (size), a boolean flag that signals if the vector owns the data (i.e. it is in charge of freeing the data), pointers to vector data on the host and the device, pointers to SUNHipExecPolicy implementations that control how the HIP kernels are launched for streaming and reduction vector kernels, and a private data structure which holds additional members that should not be accessed directly.

When instantiated with N\_VNew\_Hip, the underlying data will be allocated memory on both the host and the device. Alternatively, a user can provide host and device data arrays by using the N\_VMake\_Hip constructor. To use HIP managed memory, the constructors N\_VNewManaged\_Hip and N\_VMakeManaged\_Hip are provided. Details on each of these constructors are provided below.

To use the NVECTOR\_HIP module, the header file to include is nvector\_hip.h, and the library to link to is libsundials\_nvechip.lib. The extension .lib is typically .so for shared libraries and .a for static libraries.

### 7.10.1 NVECTOR\_HIP functions

Unlike other native SUNDIALS vector types, NVECTOR\_HIP does not provide macros to access its member variables. Instead, user should use the accessor functions:

### N\_VGetHostArrayPointer\_Hip

Prototype realtype \*N\_VGetHostArrayPointer\_Hip(N\_Vector v)

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

### ${\tt N\_VGetDeviceArrayPointer\_Hip}$

Prototype realtype \*N\_VGetDeviceArrayPointer\_Hip(N\_Vector v)

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

# $N_VIsManagedMemory_Hip$

Prototype booleantype \*N\_VIsManagedMemory\_Hip(N\_Vector v)

Description This function returns a boolean flag indicating if the vector data is allocated in managed memory or not.

The NVECTOR\_HIP module defines implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3 and 7.1.4, except for N\_VSetArrayPointer. The names of vector operations are obtained from those in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4 by appending the suffix \_Hip (e.g. N\_VDestroy\_Hip). The module NVECTOR\_HIP provides the following functions:

### N\_VNew\_Hip

Prototype N\_Vector N\_VNew\_Hip(sunindextype length)

Description This function creates an empty HIP N\_Vector with the data pointers set to NULL.

### N\_VNewManaged\_Hip

Prototype N\_Vector N\_VNewManaged\_Hip(sunindextype length)

Description This function creates and allocates memory for a HIP N\_Vector. The vector data array is allocated in managed memory.

# $\overline{\text{N}}_{\text{-}}\text{VNewEmpty\_Hip}$

Prototype N\_Vector N\_VNewEmpty\_Hip()

Description This function creates a new NVECTOR wrapper with the pointer to the wrapped HIP vector set to NULL. It is used by the N\_VNew\_Hip, N\_VMake\_Hip, and N\_VClone\_Hip implementations.

### N\_VMake\_Hip

Prototype N\_Vector N\_VMake\_Hip(sunindextype length, realtype \*h\_data, realtype \*dev\_data)

Description This function creates an NVECTOR\_HIP with user-supplied vector data arrays h\_vdata and d\_vdata. This function does not allocate memory for data itself.

# N\_VMakeManaged\_Hip

Prototype N\_Vector N\_VMakeManaged\_Hip(sunindextype length, realtype \*vdata)

Description This function creates an NVECTOR\_HIP with a user-supplied managed memory data array. This function does not allocate memory for data itself.

The module NVECTOR\_HIP also provides the following user-callable routines:

# ${\tt N\_VSetKernelExecPolicy\_Hip}$

Prototype void N\_VSetKernelExecPolicy\_Hip(N\_Vector v,

SUNHipExecPolicy\* stream\_exec\_policy,
SUNHipExecPolicy\* reduce\_exec\_policy);

Description

This function sets the execution policies which control the kernel parameters utilized when launching the streaming and reduction HIP kernels. By default the vector is setup to use the SUNHipThreadDirectExecPolicy and SUNHipBlockReduceExecPolicy. Any custom execution policy for reductions must ensure that the grid dimensions (number of thread blocks) is a multiple of the HIP warp size (64 when targeting AMD GPUs and 32 when targing NVIDIA GPUs). See section 7.10.2 below for more information about the SUNHipExecPolicy class.

Note: All vectors used in a single instance of a SUNDIALS solver must use the same execution policy. It is **strongly recommended** that this function is called immediately after constructing the vector, and any subsequent vector be created by cloning to ensure consistent execution policies across vectors.

### N\_VCopyToDevice\_Hip

Prototype void N\_VCopyToDevice\_Hip(N\_Vector v)

Description This function copies host vector data to the device.

### $N_VCopyFromDevice_Hip$

Prototype void N\_VCopyFromDevice\_Hip(N\_Vector v)

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

#### N\_VPrint\_Hip

Prototype void N\_VPrint\_Hip(N\_Vector v)

Description This function prints the content of a HIP vector to stdout.

### $N_VPrintFile_Hip$

Prototype void N\_VPrintFile\_Hip(N\_Vector v, FILE \*outfile)

Description This function prints the content of a HIP vector to outfile.

By default all fused and vector array operations are disabled in the NVECTOR\_HIP module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with N\_VNew\_Hip, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using N\_VClone. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with N\_VNew\_Hip will have the default settings for the NVECTOR\_HIP module.

# N\_VEnableFusedOps\_Hip

Prototype int N\_VEnableFusedOps\_Hip(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array op-

erations in the HIP vector. The return value is 0 for success and -1 if the input vector

or its ops structure are NULL.

### N\_VEnableLinearCombination\_Hip

Prototype int N\_VEnableLinearCombination\_Hip(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused

operation in the HIP vector. The return value is 0 for success and -1 if the input vector

or its ops structure are NULL.

### N\_VEnableScaleAddMulti\_Hip

Prototype int N\_VEnableScaleAddMulti\_Hip(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to

multiple vectors fused operation in the HIP vector. The return value is 0 for success and

-1 if the input vector or its ops structure are NULL.

### N\_VEnableDotProdMulti\_Hip

Prototype int N\_VEnableDotProdMulti\_Hip(N\_Vector v, booleantype tf)

 $\label{eq:control} \textbf{Description} \quad \textbf{This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused \\$ 

operation in the HIP vector. The return value is 0 for success and -1 if the input vector

or its ops structure are NULL.

### N\_VEnableLinearSumVectorArray\_Hip

Prototype int N\_VEnableLinearSumVectorArray\_Hip(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for

vector arrays in the HIP vector. The return value is 0 for success and -1 if the input

vector or its ops structure are NULL.

### N\_VEnableScaleVectorArray\_Hip

Prototype int N\_VEnableScaleVectorArray\_Hip(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector

arrays in the HIP vector. The return value is  $\overset{\circ}{0}$  for success and -1 if the input vector or

its ops structure are NULL.

### N\_VEnableConstVectorArray\_Hip

Prototype int N\_VEnableConstVectorArray\_Hip(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector

arrays in the HIP vector. The return value is 0 for success and -1 if the input vector or

its ops structure are NULL.

### N\_VEnableWrmsNormVectorArray\_Hip

Prototype int N\_VEnableWrmsNormVectorArray\_Hip(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the HIP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

### N\_VEnableWrmsNormMaskVectorArray\_Hip

Prototype int N\_VEnableWrmsNormMaskVectorArray\_Hip(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the HIP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

### N\_VEnableScaleAddMultiVectorArray\_Hip

Prototype int N\_VEnableScaleAddMultiVectorArray\_Hip(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the HIP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

### N\_VEnableLinearCombinationVectorArray\_Hip

Prototype int N\_VEnableLinearCombinationVectorArray\_Hip(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the HIP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

### Notes

- When there is a need to access components of an N\_Vector\_Hip, v, it is recommended to use functions N\_VGetDeviceArrayPointer\_Hip or N\_VGetHostArrayPointer\_Hip. However, when using managed memory, the function N\_VGetArrayPointer may also be used.
- To maximize efficiency, vector operations in the NVECTOR\_HIP implementation that have more than one N\_Vector argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with N\_Vector arguments that were all created with the same internal representations.

# 7.10.2 The SUNHipExecPolicy Class

In order to provide maximum flexibility to users, the HIP kernel execution parameters used by kernels within SUNDIALS are defined by objects of the sundials::HipExecPolicy abstract class type (this class can be accessed in the global namespace as SUNHipExecPolicy). Thus, users may provide custom execution policies that fit the needs of their problem. The sundials::HipExecPolicy is defined in the header file sundials\_hip\_policies.hpp, and is as follows:

```
class HipExecPolicy
{
public:
    virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const = 0;
    virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const = 0;
    virtual hipStream_t stream() const = 0;
```



```
virtual HipExecPolicy* clone() const = 0;
virtual ~HipExecPolicy() {}
};
```

To define a custom execution policy, a user simply needs to create a class that inherits from the abstract class and implements the methods. The SUNDIALS provided sundials::HipThreadDirectExecPolicy (aka in the global namespace as SUNHipThreadDirectExecPolicy) class is a good example of a what a custom execution policy may look like:

```
class HipThreadDirectExecPolicy : public HipExecPolicy
public:
  HipThreadDirectExecPolicy(const size_t blockDim, const hipStream_t stream = 0)
    : blockDim_(blockDim), stream_(stream)
  {}
  HipThreadDirectExecPolicy(const HipThreadDirectExecPolicy& ex)
   : blockDim_(ex.blockDim_), stream_(ex.stream_)
  {}
  virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const
   return (numWorkUnits + blockSize() - 1) / blockSize();
  virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const
   return blockDim_;
  virtual hipStream_t stream() const
   return stream_;
  virtual HipExecPolicy* clone() const
   return static_cast<HipExecPolicy*>(new HipThreadDirectExecPolicy(*this));
  }
private:
  const hipStream_t stream_;
  const size_t blockDim_;
};
```

In total, SUNDIALS provides 3 execution policies:

- 1. SUNHipThreadDirectExecPolicy(const size\_t blockDim, const hipStream\_t stream = 0) maps each HIP thread to a work unit. The number of threads per block (blockDim) can be set to anything. The grid size will be calculated so that there are enough threads for one thread per element. If a HIP stream is provided, it will be used to execute the kernel.
- 2. SUNHipGridStrideExecPolicy(const size\_t blockDim, const size\_t gridDim, const hipStream\_t stream = 0) is for kernels that use grid stride loops. The number of threads per block (block-Dim) can be set to anything. The number of blocks (gridDim) can be set to anything. If a HIP stream is provided, it will be used to execute the kernel.

3. SUNHipBlockReduceExecPolicy(const size\_t blockDim, const size\_t gridDim, const hipStream\_t stream = 0) is for kernels performing a reduction across individual thread blocks. The number of threads per block (blockDim) can be set to any valid multiple of the HIP warp size. The grid size (gridDim) can be set to any value greater than 0. If it is set to 0, then the grid size will be chosen so that there is enough threads for one thread per work unit. If a HIP stream is provided, it will be used to execute the kernel.

For example, a policy that uses 128 threads per block and a user provided stream can be created like so:

```
hipStream_t stream;
hipStreamCreate(&stream);
SUNHipThreadDirectExecPolicy thread_direct(128, stream);
```

These default policy objects can be reused for multiple SUNDIALS data structures since they do not hold any modifiable state information.

# 7.11 The NVECTOR\_RAJA implementation

The NVECTOR\_RAJA module is an experimental NVECTOR implementation using the RAJA hardware abstraction layer. In this implementation, RAJA allows for SUNDIALS vector kernels to run on AMD or NVIDIA 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, or the AMD ROCM HIP programming environment. When using the AMD ROCM HIP environment, the HIP-clang compiler must be utilized. Users can select which backend (CUDA or HIP) to compile with by setting the SUNDIALS\_RAJA\_BACKENDS CMake variable to either CUDA or HIP. Besides the CUDA and HIP backends, RAJA has other backends such as serial, OpenMP, and OpenACC. These backends are not used in this SUNDIALS release.

The vector content layout is as follows:

```
struct _N_VectorContent_Raja
{
   sunindextype length;
   booleantype own_helper;
   SUNMemory host_data;
   SUNMemory device_data;
   SUNMemoryHelper mem_helper;
   void* priv; /* 'private' data */
};
```

The content members are the vector length (size), a boolean flag that signals if the vector owns the memory helper, SUNMemory objects for vector data on the host and the device, a SUNMemoryHelper object and a private data structure which holds the memory management type, which should not be accessed directly.

When instantiated with N\_VNew\_Raja, the underlying data will be allocated on both the host and the device. Alternatively, a user can provide host and device data arrays by using the N\_VMake\_Raja constructor. To use managed memory, the constructors N\_VNewManaged\_Raja and

N\_VMakeManaged\_Raja are provided. Details on each of these constructors are provided below.

The header file to include when using this module is nvector\_raja.h. The installed module library to link to are libsundials\_nveccudaraja.lib when using the CUDA backend and libsundials\_nvechipraja.lib when using the HIP backend. The extension .lib is typically .so for shared libraries and .a for static libraries.

# 7.11.1 NVECTOR\_RAJA functions

Unlike other native SUNDIALS vector types, NVECTOR\_RAJA does not provide macros to access its member variables. Instead, user should use the accessor functions:

# N\_VGetHostArrayPointer\_Raja

Prototype realtype \*N\_VGetHostArrayPointer\_Raja(N\_Vector v)

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

### N\_VGetDeviceArrayPointer\_Raja

Prototype realtype \*N\_VGetDeviceArrayPointer\_Raja(N\_Vector v)

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

# N\_VSetHostArrayPointer\_Raja

Prototype realtype \*N\_VSetHostArrayPointer\_Raja(N\_Vector v)

Description This function sets the pointer to the vector data on the host. The existing pointer will

not be freed first.

### $N_VSetDeviceArrayPointer_Raja$

Prototype realtype \*N\_VSetDeviceArrayPointer\_Raja(N\_Vector v)

Description This function sets pointer to the vector data on the device. The existing pointer will

not be freed first.

### N\_VIsManagedMemory\_Raja

Prototype booleantype \*N\_VIsManagedMemory\_Raja(N\_Vector v)

Description This function returns a boolean flag indicating if the vector data is allocated in managed

memory or not.

The NVECTOR\_RAJA module defines the implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4, except for N\_VDotProdMulti, N\_VWrmsNormVectorArray, and

N\_VWrmsNormMaskVectorArray as support for arrays of reduction vectors is not yet supported in RAJA. These function will be added to the NVECTOR\_RAJA implementation in the future. Additionally the vector operations N\_VGetArrayPointer and N\_VSetArrayPointer are not provided by the RAJA vector unless managed memory is used. As such, this vector cannot be used with the SUNDIALS Fortran interfaces, nor with the SUNDIALS direct solvers and preconditioners. The NVECTOR\_RAJA module provides separate functions to access data on the host and on the device. It also provides methods for copying data from the host to the device and vice versa. Usage examples of NVECTOR\_RAJA are provided in some example programs for CVODE [35].

The names of vector operations are obtained from those in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4 by appending the suffix \_Raja (e.g. N\_VDestroy\_Raja). The module NVECTOR\_RAJA provides the following additional user-callable routines:

### N\_VNew\_Raja

Prototype N\_Vector N\_VNew\_Raja(sunindextype length)

Description This function creates and allocates memory for a RAJA N\_Vector. The vector data array

is allocated on both the host and device.

### N\_VNewWithMemHelp\_Raja

Prototype N\_Vector N\_VNewWithMemHelp\_Raja(sunindextype length, booleantype use\_managed\_mem, SUNMemoryHelper helper);

Description This function creates an NVECTOR\_RAJA which will use the SUNMemoryHelper object

to allocate memory. If use\_managed\_memory is 0, then unmanaged memory is used,

otherwise managed memory is used.

N\_VNewManaged\_Raja

Prototype N\_Vector N\_VNewManaged\_Raja(sunindextype length)

Description This function creates and allocates memory for a RAJA N\_Vector. The vector data array

is allocated in managed memory.

 $N_VNewEmpty_Raja$ 

Prototype N\_Vector N\_VNewEmpty\_Raja()

Description This function creates a new NVECTOR wrapper with the pointer to the wrapped RAJA

vector set to NULL. It is used by the N\_VNew\_Raja, N\_VMake\_Raja, and N\_VClone\_Raja

implementations.

N\_VMake\_Raja

Prototype N\_Vector N\_VMake\_Raja(sunindextype length, realtype \*h\_data, realtype \*dev\_data)

Description This function creates an NVECTOR\_RAJA with user-supplied vector data arrays h\_vdata

and d\_vdata. This function does not allocate memory for data itself.

N\_VMakeManaged\_Raja

Prototype N\_Vector N\_VMakeManaged\_Raja(sunindextype length, realtype \*vdata)

Description This function creates an NVECTOR\_RAJA with a user-supplied managed memory data

array. This function does not allocate memory for data itself.

N\_VCopyToDevice\_Raja

Prototype realtype \*N\_VCopyToDevice\_Raja(N\_Vector v)

Description This function copies host vector data to the device.

N\_VCopyFromDevice\_Raja

Prototype realtype \*N\_VCopyFromDevice\_Raja(N\_Vector v)

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

 $N_VPrint_Raja$ 

Prototype void N\_VPrint\_Raja(N\_Vector v)

Description This function prints the content of a RAJA vector to stdout.

N\_VPrintFile\_Raja

Prototype void N\_VPrintFile\_Raja(N\_Vector v, FILE \*outfile)

Description This function prints the content of a RAJA vector to outfile.

By default all fused and vector array operations are disabled in the NVECTOR\_RAJA module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with N\_VNew\_Raja, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using N\_VClone. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with N\_VNew\_Raja will have the default settings for the NVECTOR\_RAJA module.

### N\_VEnableFusedOps\_Raja

Prototype int N\_VEnableFusedOps\_Raja(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the RAJA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

### N\_VEnableLinearCombination\_Raja

Prototype int N\_VEnableLinearCombination\_Raja(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the RAJA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

# N\_VEnableScaleAddMulti\_Raja

Prototype int N\_VEnableScaleAddMulti\_Raja(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the RAJA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

### N\_VEnableLinearSumVectorArray\_Raja

Prototype int N\_VEnableLinearSumVectorArray\_Raja(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the RAJA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

# N\_VEnableScaleVectorArray\_Raja

Prototype int N\_VEnableScaleVectorArray\_Raja(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the RAJA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

# $N_{-}V$ EnableConstVectorArray\_Raja

Prototype int N\_VEnableConstVectorArray\_Raja(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the RAJA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

# N\_VEnableScaleAddMultiVectorArray\_Raja

Prototype int N\_VEnableScaleAddMultiVectorArray\_Raja(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the RAJA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N\_VEnableLinearCombinationVectorArray\_Raja

```
Prototype int N_VEnableLinearCombinationVectorArray_Raja(N_Vector v, booleantype tf)
```

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the RAJA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

#### Notes

- When there is a need to access components of an N\_Vector\_Raja, v, it is recommended to use functions N\_VGetDeviceArrayPointer\_Raja or N\_VGetHostArrayPointer\_Raja. However, when using managed memory, the function N\_VGetArrayPointer may also be used.
- To maximize efficiency, vector operations in the NVECTOR\_RAJA implementation that have more than one N\_Vector argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with N\_Vector arguments that were all created with the same internal representations.

# 7.12 The NVECTOR\_SYCL implementation

The NVECTOR\_SYCL module is an experimental NVECTOR implementation using the SYCL abstraction layer. At present the only supported SYCL compiler is the DPC++ (Intel oneAPI) compiler. This module allows for SUNDIALS vector kernels to run on Intel GPU devices. The module is intended for users who are already familiar with SYCL and GPU programming.

The vector content layout is as follows:

```
struct _N_VectorContent_Sycl
{
   sunindextype
                      length;
   booleantype
                      own_exec;
   booleantype
                      own_helper;
   SUNMemory
                      host_data;
   SUNMemory
                      device_data;
   SUNSyclExecPolicy* stream_exec_policy;
   SUNSyclExecPolicy* reduce_exec_policy;
   SUNMemoryHelper
                      mem_helper;
   sycl::queue*
                      queue;
   void*
                      priv; /* 'private' data */
};
typedef struct _N_VectorContent_Sycl *N_VectorContent_Sycl;
```

The content members are the vector length (size), boolean flags that indicate if the vector owns the execution policies and memory helper objects (i.e., it is in charge of freeing the objects), SUNMemory objects for the vector data on the host and device, pointers to execution policies that control how streaming and reduction kernels are launched, a SUNMemoryHelper for performing memory operations,



the SYCL queue, and a private data structure which holds additional members that should not be accessed directly.

When instantiated with N\_VNew\_Sycl(), the underlying data will be allocated on both the host and the device. Alternatively, a user can provide host and device data arrays by using the N\_VMake\_Sycl() constructor. To use managed (shared) memory, the constructors N\_VNewManaged\_Sycl() and N\_VMakeManaged\_Sycl() are provided. Additionally, a user-defined SUNMemoryHelper for allocating/freeing data can be provided with the constructor N\_VNewWithMemHelp\_Sycl(). Details on each of these constructors are provided below.

The header file to include when using this is nvector\_sycl.h. The installed module library to link to is libsundials\_nvecsycl.lib. The extension .lib is typically .so for shared libraries .a for static libraries.

### 7.12.1 NVECTOR\_SYCL functions

The NVECTOR\_SYCL module implementations of all vector operations listed in the sections in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4, except for N\_VDotProdMulti, N\_VWrmsNormVectorArray, and N\_VWrmsNormMaskVectorArray as support for arrays of reduction vectors is not yet supported. These function will be added to the NVECTOR\_SYCL implementation in the future. The names of vector operations are obtained from those in the aforementioned sections by appending the suffix \_Sycl (e.g., N\_VDestroy\_Sycl).

Additionally, the NVECTOR\_SYCL module provides the following user-callable constructors for creating a new NVECTOR\_SYCL:

# N\_VNew\_Sycl

Prototype N\_Vector N\_VNew\_Sycl(sunindextype length, sycl::queue\* Q)

Description This function creates and allocates memory for a SYCL N\_Vector. The vector data array is allocated on both the host and device.

### N\_VNewManaged\_Sycl

Prototype N\_Vector N\_VNewManaged\_Sycl(sunindextype length, sycl::queue\* Q)

Description This function creates and allocates memory for a SYCL N\_Vector. The vector data array is allocated in managed memory.

#### N\_VMake\_Sycl

Prototype N\_Vector N\_VMake\_Sycl(sunindextype length, realtype \*h\_data, realtype \*dev\_data, sycl::queue\* Q)

Description This function creates an NVECTOR\_SYCL with user-supplied vector data arrays h\_vdata and d\_vdata. This function does not allocate memory for data itself.

### N\_VMakeManaged\_Sycl

Prototype N\_Vector N\_VMakeManaged\_Sycl(sunindextype length, realtype \*vdata, sycl::queue\* Q)

Description This function creates an NVECTOR\_SYCL with a user-supplied managed memory data array. This function does not allocate memory for data itself.

# $\overline{N_{V}}$ NewWithMemHelp\_Sycl

Prototype N\_Vector N\_VNewWithMemHelp\_Sycl(sunindextype length,

booleantype use\_managed\_mem,

SUNMemoryHelper helper, sycl::queue\* Q);

Description This function creates an NVECTOR\_SYCL which will use the SUNMemoryHelper object

to allocate memory. If use\_managed\_memory is 0, then unmanaged memory is used,

otherwise managed memory is used.

### N\_VNewEmpty\_Sycl

Prototype N\_Vector N\_VNewEmpty\_Sycl()

Description This function creates a new NVECTOR\_SYCL where the members of the content structure

have not been allocated. This utility function is used by the other constructors to create

a new vector.

The following user-callable functions are provided for accessing the vector data arrays on the host and device and copying data between the two memory spaces. Note the generic NVECTOR operations N\_VGetArrayPointer() and N\_VSetArrayPointer() are mapped to the corresponding HostArray functions given below. To ensure memory coherency, a user will need to call the CopyTo or CopyFrom functions as necessary to transfer data between the host and device, unless managed (shared) memory is used.

### N\_VGetHostArrayPointer\_Sycl

Prototype realtype \*N\_VGetHostArrayPointer\_Sycl(N\_Vector v)

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

### N\_VGetDeviceArrayPointer\_Sycl

Prototype realtype \*N\_VGetDeviceArrayPointer\_Sycl(N\_Vector v)

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

### N\_VSetHostArrayPointer\_Sycl

Prototype realtype \*N\_VSetHostArrayPointer\_Sycl(N\_Vector v)

Description This function sets the pointer to the vector data on the host. The existing pointer will

not be freed first.

### N\_VSetDeviceArrayPointer\_Sycl

Prototype realtype \*N\_VSetDeviceArrayPointer\_Sycl(N\_Vector v)

Description This function sets pointer to the vector data on the device. The existing pointer will

not be freed first.

### N\_VCopyToDevice\_Sycl

Prototype realtype \*N\_VCopyToDevice\_Sycl(N\_Vector v)

Description This function copies host vector data to the device.

### N\_VCopyFromDevice\_Sycl

Prototype realtype \*N\_VCopyFromDevice\_Sycl(N\_Vector v)

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

# N\_VIsManagedMemory\_Sycl

Prototype booleantype \*N\_VIsManagedMemory\_Sycl(N\_Vector v)

Description This function returns a boolean flag indicating if the vector data is allocated in managed memory or not.

The following user-callable function is provided to set the execution policies for how SYCL kernels are launched on a device.

# N\_VSetKernelExecPolicy\_Sycl

Prototype int N\_VSetKernelExecPolicy\_Sycl(N\_Vector v,

SUNSyclExecPolicy \*stream\_exec\_policy,
SUNSyclExecPolicy \*reduce\_exec\_policy)

Description

This function sets the execution policies which control the kernel parameters utilized when launching the streaming and reduction kernels. By default the vector is setup to use the SUNSyclThreadDirectExecPolicy and SUNSyclBlockReduceExecPolicy. See Section 7.12.2 below for more information about the SUNSyclExecPolicy class.

**Note:** All vectors used in a single instance of a SUNDIALS package must use the same execution policy. It is **strongly recommended** that this function is called immediately after constructing the vector, and any subsequent vector be created by cloning to ensure consistent execution policies across vectors.

The following user-callable functions are provided to print the host vector data array. Unless managed memory is used, a user may need to call N\_VCopyFromDevice\_Sycl() to ensure consistency between the host and device array.

# N\_VPrint\_Sycl

Prototype void N\_VPrint\_Sycl(N\_Vector v)

Description This function prints the host data of a SYCL vector to stdout.

### N\_VPrintFile\_Sycl

Prototype void N\_VPrintFile\_Sycl(N\_Vector v, FILE \*outfile)

Description This function prints the host data of a SYCL vector to outfile.

By default all fused and vector array operations are disabled in the NVECTOR\_SYCL module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with one of the above constructors, enable/disable the desired operations on that vector with the functions below, and then use this vector in conjunction N\_VClone to create any additional vectors. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created by any of the above constructors will have the default settings for the NVECTOR\_SYCL module.

# N\_VEnableFusedOps\_Sycl

Prototype int N\_VEnableFusedOps\_Sycl(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the SYCL vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

### N\_VEnableLinearCombination\_Sycl

Prototype int N\_VEnableLinearCombination\_Sycl(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the SYCL vector. The return value is 0 for success and -1 if the input

vector or its ops structure are NULL.

### N\_VEnableScaleAddMulti\_Sycl

Prototype int N\_VEnableScaleAddMulti\_Sycl(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to

multiple vectors fused operation in the SYCL vector. The return value is 0 for success

and -1 if the input vector or its ops structure are NULL.

### N\_VEnableLinearSumVectorArray\_Sycl

Prototype int N\_VEnableLinearSumVectorArray\_Sycl(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for

vector arrays in the SYCL vector. The return value is 0 for success and -1 if the input

vector or its ops structure are NULL.

### N\_VEnableScaleVectorArray\_Sycl

Prototype int N\_VEnableScaleVectorArray\_Sycl(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector

arrays in the SYCL vector. The return value is 0 for success and -1 if the input vector

or its ops structure are NULL.

### N\_VEnableConstVectorArray\_Sycl

Prototype int N\_VEnableConstVectorArray\_Sycl(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector

arrays in the SYCL vector. The return value is 0 for success and -1 if the input vector

or its ops structure are NULL.

#### N\_VEnableScaleAddMultiVectorArray\_Sycl

Prototype int N\_VEnableScaleAddMultiVectorArray\_Sycl(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array

to multiple vector arrays operation in the  ${\tt SYCL}$  vector. The return value is  ${\tt O}$  for success

and  ${\tt -1}$  if the input vector or its  ${\tt ops}$  structure are NULL.

### ${\tt N\_VEnableLinearCombinationVectorArray\_Sycl}$

Prototype int N\_VEnableLinearCombinationVectorArray\_Sycl(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation

for vector arrays in the SYCL vector. The return value is 0 for success and -1 if the input

vector or its ops structure are NULL.

### Notes

- When there is a need to access components of an N\_Vector\_Sycl, v, it is recommended to use N\_VGetDeviceArrayPointer to access the device array or N\_VGetArrayPointer for the host array. When using managed (shared) memory, either function may be used. To ensure memory coherency, a user may need to call the CopyTo or CopyFrom functions as necessary to transfer data between the host and device, unless managed (shared) memory is used.
- To maximize efficiency, vector operations in the NVECTOR\_SYCL implementation that have more than one N\_Vector argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with N\_Vector arguments that were all created with the same internal representations.

# 7.12.2 The SUNSyclExecPolicy Class

In order to provide maximum flexibility to users, the SYCL kernel execution parameters used by kernels within SUNDIALS are defined by objects of the sundials::SyclExecPolicy abstract class type (this class can be accessed in the global namespace as SUNSyclExecPolicy). Thus, users may provide custom execution policies that fit the needs of their problem. The sundials::SyclExecPolicy is defined in the header file sundials\_sycl\_policies.hpp, as follows:

```
class SyclExecPolicy
{
public:
    virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const = 0;
    virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const = 0;
    virtual SyclExecPolicy* clone() const = 0;
    virtual ~SyclExecPolicy() {}
};
```

For consistency the function names and behavior mirror the execution policies for the CUDA and HIP vectors. In the SYCL case the blockSize is the local work-group range in a one-dimensional nd\_range (threads per group). The gridSize is the number of local work groups so the global work-group range in a one-dimensional nd\_range is blockSize \* gridSize (total number of threads). All vector kernels are written with a many-to-one mapping where work units (vector elements) are mapped in a round-robin manner across the global range. As such, the blockSize and gridSize can be set to any positive value.

To define a custom execution policy, a user simply needs to create a class that inherits from the abstract class and implements the methods. The SUNDIALS provided

sundials::SyclThreadDirectExecPolicy (aka in the global namespace as

SUNSyclThreadDirectExecPolicy) class is a good example of a what a custom execution policy may look like:



```
return (numWorkUnits + blockSize() - 1) / blockSize();
}

virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const
{
   return blockDim_;
}

virtual SyclExecPolicy* clone() const
{
   return static_cast<SyclExecPolicy*>(new SyclThreadDirectExecPolicy(*this));
}

private:
   const size_t blockDim_;
};
```

SUNDIALS provides the following execution policies:

- 1. SUNSyclThreadDirectExecPolicy(const size\_t blockDim) is for kernels performing streaming operations and maps each work unit (vector element) to a work-item (thread). Based on the local work-group range (number of threads per group, blockSize) the number of local work-groups (gridSize) is computed so there are enough work-items in the global work-group range (total number of threads, blockSize \* gridSize) for one work unit per work-item (thread).
- 2. SUNSyclGridStrideExecPolicy(const size\_t blockDim, const size\_t gridDim) is for kernels performing streaming operations and maps each work unit (vector element) to a work-item (thread) in a round-robin manner so the local work-group range (number of threads per group, blockSize) and the number of local work-groups (gridSize) can be set to any positive value. In this case the global work-group range (total number of threads, blockSize \* gridSize) may be less than the number of work units (vector elements).
- 3. SUNSyclBlockReduceExecPolicy(const size\_t blockDim) is for kernels performing a reduction, the local work-group range (number of threads per group, blockSize) and the number of local work-groups (gridSize) can be set to any positive value or the gridSize may be set to 0 in which case the global range is chosen so that there are enough threads for at most two work units per work-item.

By default the NVECTOR\_SYCL module uses the SUNSyclThreadDirectExecPolicy and SUNSyclBlockReduceExecPolicy where the default blockDim is determined by querying the device for the max\_work\_group\_size. User may specify different policies by constructing a new SyclExecPolicy and attaching it with N\_VSetKernelExecPolicy\_Sycl(). For example, a policy that uses 128 workitems (threads) per group can be created and attached like so:

```
N_Vector v = N_VNew_Sycl(length);
SUNSyclThreadDirectExecPolicy thread_direct(128);
SUNSyclBlockReduceExecPolicy block_reduce(128);
flag = N_VSetKernelExecPolicy_Sycl(v, &thread_direct, &block_reduce);
```

These default policy objects can be reused for multiple SUNDIALS data structures (e.g. a SUNMatrix and an N\_Vector) since they do not hold any modifiable state information.

# 7.13 The NVECTOR\_OPENMPDEV implementation

In situations where a user has access to a device such as a GPU for offloading computation, SUNDIALS provides an NVECTOR implementation using OpenMP device offloading, called NVECTOR\_OPENMPDEV.

The NVECTOR\_OPENMPDEV implementation defines the *content* field of the N\_Vector to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array on the host, a pointer to the beginning of a contiguous data array on the device, and a boolean flag own\_data which specifies the ownership of host and device data arrays.

```
struct _N_VectorContent_OpenMPDEV {
   sunindextype length;
   booleantype own_data;
   realtype *host_data;
   realtype *dev_data;
};
```

The header file to include when using this module is nvector\_openmpdev.h. The installed module library to link to is libsundials\_nvecopenmpdev.lib where .lib is typically .so for shared libraries and .a for static libraries.

### 7.13.1 NVECTOR\_OPENMPDEV accessor macros

The following macros are provided to access the content of an NVECTOR\_OPENMPDEV vector.

• NV\_CONTENT\_OMPDEV

This routine gives access to the contents of the NVECTOR\_OPENMPDEV vector N\_Vector.

The assignment v\_cont = NV\_CONTENT\_OMPDEV(v) sets v\_cont to be a pointer to the NVECTOR\_OPENMPDEV N\_Vector content structure.

Implementation:

```
#define NV_CONTENT_OMPDEV(v) ( (N_VectorContent_OpenMPDEV)(v->content) )
```

• NV\_OWN\_DATA\_OMPDEV, NV\_DATA\_HOST\_OMPDEV, NV\_DATA\_DEV\_OMPDEV, NV\_LENGTH\_OMPDEV

These macros give individual access to the parts of the content of an  $NVECTOR\_OPENMPDEV$   $N\_Vector$ .

The assignment  $v_{data} = NV_DATA_HOST_OMPDEV(v)$  sets  $v_{data}$  to be a pointer to the first component of the data on the host for the  $N_Vector v$ . The assignment  $NV_DATA_HOST_OMPDEV(v) = v_{data}$  sets the host component array of v to be  $v_{data}$  by storing the pointer  $v_{data}$ .

The assignment v\_dev\_data = NV\_DATA\_DEV\_OMPDEV(v) sets v\_dev\_data to be a pointer to the first component of the data on the device for the N\_Vector v. The assignment NV\_DATA\_DEV\_OMPDEV(v) = v\_dev\_data sets the device component array of v to be v\_dev\_data by storing the pointer v\_dev\_data.

The assignment  $v_len = NV_LENGTH_OMPDEV(v)$  sets  $v_len$  to be the length of v. On the other hand, the call  $NV_LENGTH_OMPDEV(v) = len_v$  sets the length of v to be  $len_v$ .

Implementation:

```
#define NV_OWN_DATA_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->own_data )
#define NV_DATA_HOST_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->host_data )
#define NV_DATA_DEV_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->dev_data )
#define NV_LENGTH_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->length )
```

### 7.13.2 NVECTOR\_OPENMPDEV functions

The NVECTOR\_OPENMPDEV module defines OpenMP device offloading implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4, except for N\_VGetArrayPointer and N\_VSetArrayPointer. As such, this vector cannot be used with the SUNDIALS Fortran interfaces, nor with the SUNDIALS direct solvers and preconditioners. It also provides methods for copying from the host to the device and vice versa.

The names of vector operations are obtained from those in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4 by appending the suffix \_OpenMPDEV (e.g. N\_VDestroy\_OpenMPDEV). The module NVECTOR\_OPENMPDEV provides the following additional user-callable routines:

### N\_VNew\_OpenMPDEV

Prototype N\_Vector N\_VNew\_OpenMPDEV(sunindextype vec\_length)

Description This function creates and allocates memory for an NVECTOR\_OPENMPDEV N\_Vector.

### N\_VNewEmpty\_OpenMPDEV

Prototype N\_Vector N\_VNewEmpty\_OpenMPDEV(sunindextype vec\_length)

Description This function creates a new NVECTOR\_OPENMPDEV N\_Vector with an empty (NULL) host and device data arrays.

# N\_VMake\_OpenMPDEV

Prototype N\_Vector N\_VMake\_OpenMPDEV(sunindextype vec\_length, realtype \*h\_vdata, realtype \*d\_vdata)

Description This function creates an NVECTOR\_OPENMPDEV vector with user-supplied vector data arrays h\_vdata and d\_vdata. This function does not allocate memory for data itself.

# | N\_VCloneVectorArray\_OpenMPDEV

Prototype N\_Vector \*N\_VCloneVectorArray\_OpenMPDEV(int count, N\_Vector w)

Description This function creates (by cloning) an array of count NVECTOR\_OPENMPDEV vectors.

### | N\_VCloneVectorArrayEmpty\_OpenMPDEV

Prototype N\_Vector \*N\_VCloneVectorArrayEmpty\_OpenMPDEV(int count, N\_Vector w)

Description This function creates (by cloning) an array of count NVECTOR\_OPENMPDEV vectors, each with an empty (NULL) data array.

### N\_VDestroyVectorArray\_OpenMPDEV

Prototype void N\_VDestroyVectorArray\_OpenMPDEV(N\_Vector \*vs, int count)

Description This function frees memory allocated for the array of count variables of type N\_Vector created with N\_VCloneVectorArray\_OpenMPDEV or with

N\_VCloneVectorArrayEmpty\_OpenMPDEV.

### N\_VGetHostArrayPointer\_OpenMPDEV

Prototype realtype \*N\_VGetHostArrayPointer\_OpenMPDEV(N\_Vector v)

Description This function returns a pointer to the host data array.

### N\_VGetDeviceArrayPointer\_OpenMPDEV

Prototype realtype \*N\_VGetDeviceArrayPointer\_OpenMPDEV(N\_Vector v)

Description This function returns a pointer to the device data array.

### N\_VPrint\_OpenMPDEV

Prototype void N\_VPrint\_OpenMPDEV(N\_Vector v)

Description This function prints the content of an NVECTOR\_OPENMPDEV vector to stdout.

### N\_VPrintFile\_OpenMPDEV

Prototype void N\_VPrintFile\_OpenMPDEV(N\_Vector v, FILE \*outfile)

Description This function prints the content of an NVECTOR\_OPENMPDEV vector to outfile.

# N\_VCopyToDevice\_OpenMPDEV

Prototype void N\_VCopyToDevice\_OpenMPDEV(N\_Vector v)

Description This function copies the content of an NVECTOR\_OPENMPDEV vector's host data array

to the device data array.

### N\_VCopyFromDevice\_OpenMPDEV

Prototype void N\_VCopyFromDevice\_OpenMPDEV(N\_Vector v)

Description This function copies the content of an NVECTOR\_OPENMPDEV vector's device data array

to the host data array.

By default all fused and vector array operations are disabled in the NVECTOR\_OPENMPDEV module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with N\_VNew\_OpenMPDEV, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using N\_VClone. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with N\_VNew\_OpenMPDEV will have the default settings for the NVECTOR\_OPENMPDEV module.

### N\_VEnableFusedOps\_OpenMPDEV

Prototype int N\_VEnableFusedOps\_OpenMPDEV(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array op-

erations in the  ${\tt NVECTOR\_OPENMPDEV}$  vector. The return value is 0 for success and -1

if the input vector or its ops structure are NULL.

# N\_VEnableLinearCombination\_OpenMPDEV

Prototype int N\_VEnableLinearCombination\_OpenMPDEV(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused

operation in the NVECTOR\_OPENMPDEV vector. The return value is 0 for success and

-1 if the input vector or its ops structure are NULL.

### ${\tt N\_VEnableScaleAddMulti\_OpenMPDEV}$

Prototype int N\_VEnableScaleAddMulti\_OpenMPDEV(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to

multiple vectors fused operation in the NVECTOR\_OPENMPDEV vector. The return value

is 0 for success and -1 if the input vector or its ops structure are NULL.

# N\_VEnableDotProdMulti\_OpenMPDEV

Prototype int N\_VEnableDotProdMulti\_OpenMPDEV(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the NVECTOR\_OPENMPDEV vector. The return value is 0 for success and

-1 if the input vector or its ops structure are NULL.

### N\_VEnableLinearSumVectorArray\_OpenMPDEV

Prototype int N\_VEnableLinearSumVectorArray\_OpenMPDEV(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the NVECTOR\_OPENMPDEV vector. The return value is 0 for success

and -1 if the input vector or its ops structure are NULL.

### N\_VEnableScaleVectorArray\_OpenMPDEV

Prototype int N\_VEnableScaleVectorArray\_OpenMPDEV(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector

arrays in the NVECTOR\_OPENMPDEV vector. The return value is 0 for success and -1 if

the input vector or its ops structure are NULL.

# N\_VEnableConstVectorArray\_OpenMPDEV

Prototype int N\_VEnableConstVectorArray\_OpenMPDEV(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector

arrays in the NVECTOR\_OPENMPDEV vector. The return value is 0 for success and -1 if

the input vector or its ops structure are NULL.

### N\_VEnableWrmsNormVectorArray\_OpenMPDEV

Prototype int N\_VEnableWrmsNormVectorArray\_OpenMPDEV(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for

vector arrays in the  ${\tt NVECTOR\_OPENMPDEV}$  vector. The return value is 0 for success

and -1 if the input vector or its ops structure are NULL.

### N\_VEnableWrmsNormMaskVectorArray\_OpenMPDEV

Prototype int N\_VEnableWrmsNormMaskVectorArray\_OpenMPDEV(N\_Vector v,

booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm op-

eration for vector arrays in the NVECTOR\_OPENMPDEV vector. The return value is 0 for

success and -1 if the input vector or its ops structure are NULL.

### N\_VEnableScaleAddMultiVectorArray\_OpenMPDEV

Prototype int N\_VEnableScaleAddMultiVectorArray\_OpenMPDEV(N\_Vector v,

booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array

to multiple vector arrays operation in the  ${\tt NVECTOR\_OPENMPDEV}$  vector. The return

value is 0 for success and -1 if the input vector or its ops structure are NULL.

# | N\_VEnableLinearCombinationVectorArray\_OpenMPDEV

Prototype int N\_VEnableLinearCombinationVectorArray\_OpenMPDEV(N\_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the NVECTOR\_OPENMPDEV vector. The return value is 0 for success

and -1 if the input vector or its ops structure are NULL.

### Notes

• When looping over the components of an N\_Vector v, it is most efficient to first obtain the component array via h\_data = NV\_DATA\_HOST\_OMPDEV(v) for the host array or d\_data = NV\_DATA\_DEV\_OMPDEV(v) for the device array and then access h\_data[i] or d\_data[i] within the loop.

- When accessing individual components of an N\_Vector v on the host remember to first copy the array back from the device with N\_VCopyFromDevice\_OpenMPDEV(v) to ensure the array is up to date.
- N\_VNewEmpty\_OpenMPDEV, N\_VMake\_OpenMPDEV, and N\_VCloneVectorArrayEmpty\_OpenMPDEV set the field own\_data = SUNFALSE. N\_VDestroy\_OpenMPDEV and N\_VDestroyVectorArray\_OpenMPDEV will not attempt to free the pointer data for any N\_Vector with own\_data set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the data pointer.
- To maximize efficiency, vector operations in the NVECTOR\_OPENMPDEV implementation that have more than one N\_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N\_Vector arguments that were all created with the same internal representations.

# 7.14 The NVECTOR\_TRILINOS implementation

The NVECTOR\_TRILINOS module is an NVECTOR wrapper around the Trilinos Tpetra vector. The interface to Tpetra is implemented in the Sundials::TpetraVectorInterface class. This class simply stores a reference counting pointer to a Tpetra vector and inherits from an empty structure

```
struct _N_VectorContent_Trilinos {};
```

to interface the C++ class with the NVECTOR C code. A pointer to an instance of this class is kept in the content field of the N\_Vector object, to ensure that the Tpetra vector is not deleted for as long as the N\_Vector object exists.

The Tpetra vector type in the Sundials::TpetraVectorInterface class is defined as:

```
typedef Tpetra::Vector<realtype, int, sunindextype> vector_type;
```

The Tpetra vector will use the SUNDIALS-specified realtype as its scalar type, int as its local ordinal type, and sunindextype as the global ordinal type. This type definition will use Tpetra's default node type. Available Kokkos node types in Trilinos 12.14 release are serial (single thread), OpenMP, Pthread, and CUDA. The default node type is selected when building the Kokkos package. For example, the Tpetra vector will use a CUDA node if Tpetra was built with CUDA support and the CUDA node was selected as the default when Tpetra was built.

The header file to include when using this module is nvector\_trilinos.h. The installed module library to link to is libsundials\_nvectrilinos.lib where .lib is typically .so for shared libraries and .a for static libraries.





### 7.14.1 NVECTOR\_TRILINOS functions

The NVECTOR\_TRILINOS module defines implementations of all vector operations listed in Tables 7.1.1, 7.1.4, and 7.1.4, except for N\_VGetArrayPointer and N\_VSetArrayPointer. As such, this vector cannot be used with SUNDIALS Fortran interfaces, nor with the SUNDIALS direct solvers and preconditioners. When access to raw vector data is needed, it is recommended to extract the Trilinos Tpetra vector first, and then use Tpetra vector methods to access the data. Usage examples of NVECTOR\_TRILINOS are provided in example programs for IDA [34].

The names of vector operations are obtained from those in Tables 7.1.1, 7.1.4, and 7.1.4 by appending the suffix \_Trilinos (e.g. N\_VDestroy\_Trilinos). Vector operations call existing Tpetra::Vector methods when available. Vector operations specific to SUNDIALS are implemented as standalone functions in the namespace Sundials::TpetraVector, located in the file SundialsTpetraVectorKernels.hpp. The module NVECTOR\_TRILINOS provides the following additional user-callable functions:

#### • N\_VGetVector\_Trilinos

This C++ function takes an N\_Vector as the argument and returns a reference counting pointer to the underlying Tpetra vector. This is a standalone function defined in the global namespace.

```
Teuchos::RCP<vector_type> N_VGetVector_Trilinos(N_Vector v);
```

### • N\_VMake\_Trilinos

This C++ function creates and allocates memory for an NVECTOR\_TRILINOS wrapper around a user-provided Tpetra vector. This is a standalone function defined in the global namespace.

```
N_Vector N_VMake_Trilinos(Teuchos::RCP<vector_type> v);
```

### Notes

- The template parameter vector\_type should be set as:
   typedef Sundials::TpetraVectorInterface::vector\_type vector\_type
   This will ensure that data types used in Tpetra vector match those in SUNDIALS.
- When there is a need to access components of an N\_Vector\_Trilinos, v, it is 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.

# 7.15 The NVECTOR\_MANYVECTOR implementation

The NVECTOR\_MANYVECTOR implementation of the NVECTOR module provided with SUNDIALS is designed to facilitate problems with an inherent data partitioning for the solution vector within a computational node. These data partitions are entirely user-defined, through construction of distinct NVECTOR modules for each component, that are then combined together to form the NVECTOR\_MANYVECTOR. We envision two generic use cases for this implementation:

A. Heterogeneous computational architectures: for users who wish to partition data on a node between different computing resources, they may create architecture-specific subvectors for each partition. For example, a user could create one serial component based on NVECTOR\_SERIAL, another component for GPU accelerators based on NVECTOR\_CUDA, and another threaded component based on NVECTOR\_OPENMP.

B. Structure of arrays (SOA) data layouts: for users who wish to create separate subvectors for each solution component, e.g., in a Navier-Stokes simulation they could have separate subvectors for density, velocities and pressure, which are combined together into a single NVECTOR\_MANYVECTOR for the overall "solution".

We note that the above use cases are not 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 required set of operations. 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, DAE residual, or nonlinear solver residual functions.

### 7.15.1 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.

### 7.15.2 NVECTOR\_MANYVECTOR functions

The NVECTOR\_MANYVECTOR module implements all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4, except for N\_VGetArrayPointer, N\_VSetArrayPointer, N\_VScaleAddMultiVectorArray, and N\_VLinearCombinationVectorArray. As such, this vector cannot be used with the SUNDIALS Fortran-77 interfaces, nor with the SUNDIALS direct solvers and preconditioners. Instead, the NVECTOR\_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 Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4 by appending the suffix \_ManyVector (e.g. N\_VDestroy\_ManyVector). The module NVECTOR\_MANYVECTOR provides the following additional user-callable routines:

```
N_VNew_ManyVector
```

Description This function creates a ManyVector from a set of existing NVECTOR objects.

This routine will copy all N\_Vector pointers from the input vec\_array, so the user may modify/free that pointer array after calling this function. However, this routine does not allocate any new subvectors, so the underlying NVECTOR objects themselves should not be destroyed before the ManyVector that contains them.

Upon successful completion, the new ManyVector is returned; otherwise this routine returns NULL (e.g., a memory allocation failure occurred).

Users of the Fortran 2003 interface to this function will first need to use the generic N\_Vector utility functions N\_VNewVectorArray, and N\_VSetVecAtIndexVectorArray to create the N\_Vector\* argument. This is further explained in Chapter 5.1.3.5, and the functions are documented in Chapter 7.1.6.

F2003 Name This function is callable as FN\_VNew\_ManyVector when using the Fortran 2003 interface module.

# N\_VGetSubvector\_ManyVector

Prototype N\_Vector N\_VGetSubvector\_ManyVector(N\_Vector v, sunindextype vec\_num);

Description This function returns the vec\_num subvector from the NVECTOR array.

F2003 Name This function is callable as FN\_VGetSubvector\_ManyVector when using the Fortran 2003 interface module.

### N\_VGetSubvectorArrayPointer\_ManyVector

Prototype realtype \*N\_VGetSubvectorArrayPointer\_ManyVector(N\_Vector v, sunindextype vec\_num);

Description This function returns the data array pointer for the vec\_num subvector from the NVEC-TOR array.

If the input vec\_num is invalid, or if the subvector does not support the N\_VGetArrayPointer operation, then NULL is returned.

F2003 Name This function is callable as FN\_VGetSubvectorArrayPointer\_ManyVector when using the Fortran 2003 interface module.

### N\_VSetSubvectorArrayPointer\_ManyVector

Prototype int N\_VSetSubvectorArrayPointer\_ManyVector(realtype \*v\_data, N\_Vector v, sunindextype vec\_num);

Description This function sets the data array pointer for the vec\_num subvector from the NVECTOR array.

If the input vec\_num is invalid, or if the subvector does not support the N\_VSetArrayPointer operation, then this routine returns -1; otherwise it returns 0.

F2003 Name This function is callable as FN\_VSetSubvectorArrayPointer\_ManyVector when using the Fortran 2003 interface module.

### N\_VGetNumSubvectors\_ManyVector

Prototype sunindextype N\_VGetNumSubvectors\_ManyVector(N\_Vector v);

Description This function returns the overall number of subvectors in the ManyVector object.

F2003 Name This function is callable as FN\_VGetNumSubvectors\_ManyVector when using the Fortran 2003 interface module.

By default all fused and vector array operations are disabled in the NVECTOR\_MANYVECTOR module, except for N\_VWrmsNormVectorArray and N\_VWrmsNormMaskVectorArray, that are enabled by default. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with N\_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.

### N\_VEnableFusedOps\_ManyVector

Prototype int N\_VEnableFusedOps\_ManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableFusedOps\_ManyVector when using the Fortran 2003 interface module.

# ${\tt N\_VEnableLinearCombination\_ManyVector}$

Prototype int N\_VEnableLinearCombination\_ManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableLinearCombination\_ManyVector when using the Fortran 2003 interface module.

### N\_VEnableScaleAddMulti\_ManyVector

Prototype int N\_VEnableScaleAddMulti\_ManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableScaleAddMulti\_ManyVector when using the Fortran 2003 interface module.

# ${\tt N\_VEnableDotProdMulti\_ManyVector}$

Prototype int N\_VEnableDotProdMulti\_ManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableDotProdMulti\_ManyVector when using the Fortran 2003 interface module.

# N\_VEnableLinearSumVectorArray\_ManyVector

Prototype int N\_VEnableLinearSumVectorArray\_ManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableLinearSumVectorArray\_ManyVector when using the Fortran 2003 interface module.

### N\_VEnableScaleVectorArray\_ManyVector

Prototype int N\_VEnableScaleVectorArray\_ManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableScaleVectorArray\_ManyVector when using the Fortran 2003 interface module.

### N\_VEnableConstVectorArray\_ManyVector

Prototype int N\_VEnableConstVectorArray\_ManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableConstVectorArray\_ManyVector when using the Fortran 2003 interface module.

### N\_VEnableWrmsNormVectorArray\_ManyVector

Prototype int N\_VEnableWrmsNormVectorArray\_ManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableWrmsNormVectorArray\_ManyVector when using the Fortran 2003 interface module.

### N\_VEnableWrmsNormMaskVectorArray\_ManyVector

Prototype int N\_VEnableWrmsNormMaskVectorArray\_ManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the ManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableWrmsNormMaskVectorArray\_ManyVector when using the Fortran 2003 interface module.

#### Notes

• N\_VNew\_ManyVector sets the field own\_data = SUNFALSE. N\_VDestroy\_ManyVector will not attempt to call N\_VDestroy on any subvectors contained in the subvector array for any N\_Vector with own\_data set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the subvectors.





• To maximize efficiency, arithmetic vector operations in the NVECTOR\_MANYVECTOR implementation that have more than one N\_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N\_Vector arguments that were all created with the same subvector representations.

# 7.16 The NVECTOR\_MPIMANYVECTOR implementation

The NVECTOR\_MPIMANYVECTOR implementation of the NVECTOR module provided with SUNDIALS is designed to facilitate problems with an inherent data partitioning for the solution vector, and when using distributed-memory parallel architectures. As such, the MPIManyVector implementation supports all use cases allowed by the MPI-unaware ManyVector implementation, as well as partitioning data between nodes in a parallel environment. These data partitions are entirely user-defined, through construction of distinct NVECTOR modules for each component, that are then combined together to form the NVECTOR\_MPIMANYVECTOR. We envision three generic use cases for this implementation:

- A. Heterogeneous computational architectures (single-node or multi-node): for users who wish to partition data on a node between different computing resources, they may create architecture-specific subvectors for each partition. For example, a user could create one MPI-parallel component based on NVECTOR\_PARALLEL, another single-node component for GPU accelerators based on NVECTOR\_CUDA, and another threaded single-node component based on NVECTOR\_OPENMP.
- B. Process-based multiphysics decompositions (multi-node): for users who wish to combine separate simulations together, e.g., where one subvector resides on one subset of MPI processes, while another subvector resides on a different subset of MPI processes, and where the user has created a MPI intercommunicator to connect these distinct process sets together.
- C. Structure of arrays (SOA) data layouts (single-node or multi-node): for users who wish to create separate subvectors for each solution component, e.g., in a Navier-Stokes simulation they could have separate subvectors for density, velocities and pressure, which are combined together into a single NVECTOR\_MPIMANYVECTOR for the overall "solution".

We note that the above use cases are not mutually exclusive, and the NVECTOR\_MPIMANYVECTOR implementation should support arbitrary combinations of these cases.

The NVECTOR\_MPIMANYVECTOR implementation is designed to work with any NVECTOR subvectors that implement the minimum *required* set of operations, however significant performance benefits may be obtained when subvectors additionally implement the optional local reduction operations listed in Table 7.1.4.

Additionally, NVECTOR\_MPIMANYVECTOR sets no limit on the number of subvectors that may be attached (aside from the limitations of using sunindextype for indexing, and standard per-node memory limitations). However, while this ostensibly supports subvectors with one entry each (i.e., one subvector for each solution entry), we anticipate that this extreme situation will hinder performance due to non-stride-one memory accesses and increased function call overhead. We therefore recommend a relatively coarse partitioning of the problem, although actual performance will likely be problem-dependent.

As a final note, in the coming years we plan to introduce additional algebraic solvers and time integration modules that will leverage the problem partitioning enabled by NVECTOR\_MPIMANYVECTOR. However, even at present we anticipate that users will be able to leverage such data partitioning in their problem-defining ODE right-hand side, DAE residual, or nonlinear solver residual functions.

### 7.16.1 NVECTOR\_MPIMANYVECTOR structure

The NVECTOR\_MPIMANYVECTOR implementation defines the *content* field of N\_Vector to be a structure containing the MPI communicator (or MPI\_COMM\_NULL if running on a single-node), the number of subvectors comprising the MPIManyVector, the global length of the MPIManyVector (including all subvectors on all MPI tasks), a pointer to the beginning of the array of subvectors, and a boolean flag own\_data indicating ownership of the subvectors that populate subvec\_array.

```
struct _N_VectorContent_MPIManyVector {
                                 /* overall MPI communicator
  MPI_Comm
                                                                     */
                comm;
  sunindextype
                num_subvectors; /* number of vectors attached
  sunindextype
                                 /* overall mpimanyvector length
                global_length;
  N_Vector*
                subvec_array;
                                 /* pointer to N_Vector array
                                 /* flag indicating data ownership
  booleantype
                own_data;
};
```

The header file to include when using this module is nvector\_mpimanyvector.h. The installed module library to link against is libsundials\_nvecmpimanyvector.lib where .lib is typically .so for shared libraries and .a for static libraries.

**Note:** If SUNDIALS is configured with MPI disabled, then the MPIManyVector library will not be built. Furthermore, any user codes that include nvector\_mpimanyvector.h *must* be compiled using an MPI-aware compiler (whether the specific user code utilizes MPI or not). We note that the NVECTOR\_MANYVECTOR implementation is designed for ManyVector use cases in an MPI-unaware environment.

### 7.16.2 NVECTOR\_MPIMANYVECTOR functions

The NVECTOR\_MPIMANYVECTOR module implements all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4, except for N\_VGetArrayPointer, N\_VSetArrayPointer, N\_VScaleAddMultiVectorArray, and N\_VLinearCombinationVectorArray. As such, this vector cannot be used with the SUNDIALS Fortran-77 interfaces, nor with the SUNDIALS direct solvers and preconditioners. Instead, the NVECTOR\_MPIMANYVECTOR module provides functions to access subvectors, whose data may in turn be accessed according to their NVECTOR implementations.

The names of vector operations are obtained from those in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4 by appending the suffix \_MPIManyVector (e.g. N\_VDestroy\_MPIManyVector). The module NVECTOR\_MPIMANYVECTOR provides the following additional user-callable routines:

```
N_VNew_MPIManyVector
```

Prototype N\_Vector N\_VNew\_MPIManyVector(sunindextype num\_subvectors, N\_Vector \*vec\_array);

Description

This function creates an MPIManyVector from a set of existing NVECTOR objects, under the requirement that all MPI-aware subvectors use the same MPI communicator (this is checked internally). If none of the subvectors are MPI-aware, then this may equivalently be used to describe data partitioning within a single node. We note that this routine is designed to support use cases A and C above.

This routine will copy all N\_Vector pointers from the input vec\_array, so the user may modify/free that pointer array after calling this function. However, this routine does not allocate any new subvectors, so the underlying NVECTOR objects themselves should not be destroyed before the MPIManyVector that contains them.

Upon successful completion, the new MPIManyVector is returned; otherwise this routine returns NULL (e.g., if two MPI-aware subvectors use different MPI communicators).

Users of the Fortran 2003 interface to this function will first need to use the generic N\_Vector utility functions N\_VNewVectorArray, and N\_VSetVecAtIndexVectorArray to create the N\_Vector\* argument. This is further explained in Chapter 5.1.3.5, and the functions are documented in Chapter 7.1.6.

F2003 Name This function is callable as FN\_VNew\_MPIManyVector when using the Fortran 2003 interface module.



### N\_VMake\_MPIManyVector

Description

This function creates an MPIManyVector from a set of existing NVECTOR objects, and a user-created MPI communicator that "connects" these subvectors. Any MPI-aware subvectors may use different MPI communicators than the input comm. We note that this routine is designed to support any combination of the use cases above.

The input comm should be this user-created MPI communicator. This routine will internally call MPI\_Comm\_dup to create a copy of the input comm, so the user-supplied comm argument need not be retained after the call to N\_VMake\_MPIManyVector.

If all subvectors are MPI-unaware, then the input comm argument should be MPI\_COMM\_NULL, although in this case, it would be simpler to call N\_VNew\_MPIManyVector instead, or to just use the NVECTOR\_MANYVECTOR module.

This routine will copy all N\_Vector pointers from the input vec\_array, so the user may modify/free that pointer array after calling this function. However, this routine does not allocate any new subvectors, so the underlying NVECTOR objects themselves should not be destroyed before the MPIManyVector that contains them.

Upon successful completion, the new MPIManyVector is returned; otherwise this routine returns NULL (e.g., if the input vec\_array is NULL).

F2003 Name This function is callable as FN\_VMake\_MPIManyVector when using the Fortran 2003 interface module.

### N\_VGetSubvector\_MPIManyVector

Prototype N\_Vector N\_VGetSubvector\_MPIManyVector(N\_Vector v, sunindextype vec\_num);

Description This function returns the vec\_num subvector from the NVECTOR array.

F2003 Name This function is callable as FN\_VGetSubvector\_MPIManyVector when using the Fortran 2003 interface module.

### N\_VGetSubvectorArrayPointer\_MPIManyVector

Prototype realtype \*N\_VGetSubvectorArrayPointer\_MPIManyVector(N\_Vector v, sunindextype vec\_num);

Description This function returns the data array pointer for the vec\_num subvector from the NVECTOR array.

If the input vec\_num is invalid, or if the subvector does not support the N\_VGetArrayPointer operation, then NULL is returned.

F2003 Name This function is callable as FN\_VGetSubvectorArrayPointer\_MPIManyVector when using the Fortran 2003 interface module.

### N\_VSetSubvectorArrayPointer\_MPIManyVector

Prototype int N\_VSetSubvectorArrayPointer\_MPIManyVector(realtype \*v\_data, N\_Vector v, sunindextype vec\_num);

Description This function sets the data array pointer for the vec\_num subvector from the NVECTOR array.

If the input vec\_num is invalid, or if the subvector does not support the N\_VSetArrayPointer operation, then this routine returns -1; otherwise it returns 0.

F2003 Name This function is callable as FN\_VSetSubvectorArrayPointer\_MPIManyVector when using the Fortran 2003 interface module.

### N\_VGetNumSubvectors\_MPIManyVector

Prototype sunindextype N\_VGetNumSubvectors\_MPIManyVector(N\_Vector v);

Description This function returns the overall number of subvectors in the MPIManyVector object.

F2003 Name This function is callable as FN\_VGetNumSubvectors\_MPIManyVector when using the Fortran 2003 interface module.

By default all fused and vector array operations are disabled in the NVECTOR\_MPIMANYVECTOR module, except for N\_VWrmsNormVectorArray and N\_VWrmsNormMaskVectorArray, that are enabled by default. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with N\_VNew\_MPIManyVector or N\_VMake\_MPIManyVector, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using N\_VClone. This guarantees that the new vectors will have the same operations enabled/disabled, since cloned vectors inherit those configuration options from the vector they are cloned from, while vectors created with N\_VNew\_MPIManyVector and N\_VMake\_MPIManyVector will have the default settings for the NVECTOR\_MPIMANYVECTOR module. We note that these routines do not call the corresponding routines on subvectors, so those should be set up as desired before attaching them to the MPIManyVector in N\_VNew\_MPIManyVector or N\_VMake\_MPIManyVector.

### N\_VEnableFusedOps\_MPIManyVector

Prototype int N\_VEnableFusedOps\_MPIManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the MPIManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableFusedOps\_MPIManyVector when using the Fortran 2003 interface module.

### N\_VEnableLinearCombination\_MPIManyVector

Prototype int N\_VEnableLinearCombination\_MPIManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the MPIManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableLinearCombination\_MPIManyVector when using the Fortran 2003 interface module.

### N\_VEnableScaleAddMulti\_MPIManyVector

Prototype int N\_VEnableScaleAddMulti\_MPIManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the MPIManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableScaleAddMulti\_MPIManyVector when using the Fortran 2003 interface module.

# N\_VEnableDotProdMulti\_MPIManyVector

Prototype int N\_VEnableDotProdMulti\_MPIManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the MPIManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableDotProdMulti\_MPIManyVector when using the Fortran 2003 interface module.

# N\_VEnableLinearSumVectorArray\_MPIManyVector

Prototype int N\_VEnableLinearSumVectorArray\_MPIManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the MPIManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableLinearSumVectorArray\_MPIManyVector when using the Fortran 2003 interface module.

### N\_VEnableScaleVectorArray\_MPIManyVector

Prototype int N\_VEnableScaleVectorArray\_MPIManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the MPIManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableScaleVectorArray\_MPIManyVector when using the Fortran 2003 interface module.

### N\_VEnableConstVectorArray\_MPIManyVector

Prototype int N\_VEnableConstVectorArray\_MPIManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the MPIManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableConstVectorArray\_MPIManyVector when using the Fortran 2003 interface module.

# N\_VEnableWrmsNormVectorArray\_MPIManyVector

Prototype int N\_VEnableWrmsNormVectorArray\_MPIManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the MPIManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableWrmsNormVectorArray\_MPIManyVector when using the Fortran 2003 interface module.

# N\_VEnableWrmsNormMaskVectorArray\_MPIManyVector

Prototype int N\_VEnableWrmsNormMaskVectorArray\_MPIManyVector(N\_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the MPIManyVector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN\_VEnableWrmsNormMaskVectorArray\_MPIManyVector when using the Fortran 2003 interface module.

### Notes

- N\_VNew\_MPIManyVector and N\_VMake\_MPIManyVector set the field own\_data = SUNFALSE.

  N\_VDestroy\_MPIManyVector will not attempt to call N\_VDestroy on any subvectors contained in the subvector array for any N\_Vector with own\_data set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the subvectors.
- To maximize efficiency, arithmetic vector operations in the NVECTOR\_MPIMANYVECTOR implementation that have more than one N\_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N\_Vector arguments that were all created with the same subvector representations.



# 7.17 The NVECTOR\_MPIPLUSX implementation

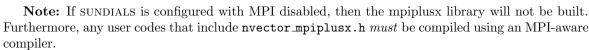
The NVECTOR\_MPIPLUSX implementation of the NVECTOR module provided with SUNDIALS is designed to facilitate the MPI+X paradigm, where X is some form of on-node (local) parallelism (e.g. OpenMP, CUDA). This paradigm is becoming increasingly popular with the rise of heterogeneous computing architectures.

The NVECTOR\_MPIPLUSX implementation is designed to work with any NVECTOR that implements the minimum required set of operations. However, it is not recommended to use the NVECTOR\_PARALLEL, NVECTOR\_PARHYP, NVECTOR\_PETSC, or NVECTOR\_TRILINOS implementations underneath the NVECTOR\_MPIPLUSX module since they already provide MPI capabilities.

### 7.17.1 NVECTOR\_MPIPLUSX structure

The NVECTOR\_MPIPLUSX implementation is a thin wrapper around the NVECTOR\_MPIMANYVECTOR. Accordingly, it adopts the same content structure as defined in Section 7.16.1.

The header file to include when using this module is nvector\_mpiplusx.h. The installed module library to link against is libsundials\_nvecmpiplusx.lib where .lib is typically .so for shared libraries and .a for static libraries.





### 7.17.2 NVECTOR MPIPLUSX functions

The NVECTOR\_MPIPLUSX module adopts all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4, from the NVECTOR\_MPIMANYVECTOR (see section 7.16.2) except for N\_VGetArrayPointer and N\_VSetArrayPointer; the module provides its own implementation of these functions that call the local vector implementations. Therefore, the NVECTOR\_MPIPLUSX module implements all of the operations listed in the referenced sections except for N\_VScaleAddMultiVectorArray, and N\_VLinearCombinationVectorArray Accordingly, it's compatibility with the SUNDIALS Fortran-77 interface, and with the SUNDIALS direct solvers and preconditioners depends on the local vector implementation.

The module NVECTOR\_MPIPLUSX provides the following additional user-callable routines:

### N\_VMake\_MPIPlusX

Prototype N\_Vector N\_VMake\_MPIPlusX(MPI\_Comm comm,

N\_Vector \*local\_vector);

Description This function creates an MPIPlusX vector from an existing local (i.e. on-node) NVECTOR object, and a user-created MPI communicator.

The input comm should be this user-created MPI communicator. This routine will internally call MPI\_Comm\_dup to create a copy of the input comm, so the user-supplied comm argument need not be retained after the call to N\_VMake\_MPIPlusX.

This routine will copy the N\_Vector pointer to the input local\_vector, so the underlying local NVECTOR object should not be destroyed before the mpiplusx that contains it.

Upon successful completion, the new MPIPlusX is returned; otherwise this routine returns NULL (e.g., if the input local\_vector is NULL).

F2003 Name This function is callable as FN\_VMake\_MPIPlusX when using the Fortran 2003 interface module.

### N\_VGetLocalVector\_MPIPlusX

N\_Vector N\_VGetLocalVector\_MPIPlusX(N\_Vector v); Prototype

This function returns the local vector underneath the MPIPlusX NVECTOR. Description

F2003 Name This function is callable as FN\_VGetLocalVector\_MPIPlusX when using the Fortran 2003 interface module.

### N\_VGetArrayPointer\_MPIPlusX

realtype\* N\_VGetLocalVector\_MPIPlusX(N\_Vector v); Prototype

This function returns the data array pointer for the local vector if the local vector Description implements the N\_VGetArrayPointer operation; otherwise it returns NULL.

F2003 Name This function is callable as FN\_VGetArrayPointer\_MPIPlusX when using the Fortran 2003 interface module.

### N\_VSetArrayPointer\_MPIPlusX

Prototype void N\_VSetArrayPointer\_MPIPlusX(realtype \*data, N\_Vector v);

This function sets the data array pointer for the local vector if the local vector imple-Description ments the N\_VSetArrayPointer operation.

F2003 Name This function is callable as FN\_VSetArrayPointer\_MPIPlusX when using the Fortran 2003 interface module.

The NVECTOR\_MPIPLUSX module does not implement any fused or vector array operations. Instead users should enable/disable fused operations on the local vector.

### Notes



- N\_VMake\_MPIPlusX sets the field  $own_{-}data = SUNFALSE$ . and N\_VDestroy\_MPIPlusX will not call N\_VDestroy on the local vector. In this case, it is the user's responsibility to deallocate the local vector.
- To maximize efficiency, arithmetic vector operations in the NVECTOR\_MPIPLUSX implementation that have more than one N\_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N\_Vector arguments that were all created with the same local vector representations.

#### 7.18 **NVECTOR** Examples

There are NVector examples that may be installed for the implementations provided with SUNDIALS. Each implementation makes use of the functions in test\_nvector.c. These example functions show simple usage of the NVector family of functions. The input to the examples are the vector length, number of threads (if threaded implementation), and a print timing flag. The following is a list of the example functions in test\_nvector.c:

• Test\_N\_VClone: Creates clone of vector and checks validity of clone.

- Test\_N\_VCloneEmpty: Creates clone of empty vector and checks validity of clone.
- Test\_N\_VCloneVectorArray: Creates clone of vector array and checks validity of cloned array.
- Test\_N\_VCloneVectorArray: Creates clone of empty vector array and checks validity of cloned array.
- Test\_N\_VGetArrayPointer: Get array pointer.
- Test\_N\_VSetArrayPointer: Allocate new vector, set pointer to new vector array, and check values.
- Test\_N\_VGetLength: Compares self-reported length to calculated length.
- Test\_N\_VGetCommunicator: Compares self-reported communicator to the one used in constructor; or for MPI-unaware vectors it ensures that NULL is reported.
- Test\_N\_VLinearSum Case 1a: Test y = x + y
- Test\_N\_VLinearSum Case 1b: Test y = -x + y
- Test\_N\_VLinearSum Case 1c: Test y = ax + y
- Test\_N\_VLinearSum Case 2a: Test x = x + y
- Test\_N\_VLinearSum Case 2b: Test x = x y
- Test\_N\_VLinearSum Case 2c: Test x = x + by
- Test\_N\_VLinearSum Case 3: Test z = x + y
- Test\_N\_VLinearSum Case 4a: Test z = x y
- Test\_N\_VLinearSum Case 4b: Test z = -x + y
- Test\_N\_VLinearSum Case 5a: Test z = x + by
- Test\_N\_VLinearSum Case 5b: Test z = ax + y
- Test\_N\_VLinearSum Case 6a: Test z = -x + by
- Test\_N\_VLinearSum Case 6b: Test z = ax y
- Test\_N\_VLinearSum Case 7: Test z = a(x + y)
- Test\_N\_VLinearSum Case 8: Test z = a(x y)
- Test\_N\_VLinearSum Case 9: Test z = ax + by
- Test\_N\_VConst: Fill vector with constant and check result.
- Test\_N\_VProd: Test vector multiply: z = x \* y
- Test\_N\_VDiv: Test vector division: z = x / y
- Test\_N\_VScale: Case 1: scale: x = cx
- Test\_N\_VScale: Case 2: copy: z = x
- Test\_N\_VScale: Case 3: negate: z = -x
- Test\_N\_VScale: Case 4: combination: z = cx
- Test\_N\_VAbs: Create absolute value of vector.

- Test\_N\_VAddConst: add constant vector: z = c + x
- Test\_N\_VDotProd: Calculate dot product of two vectors.
- Test\_N\_VMaxNorm: Create vector with known values, find and validate the max norm.
- Test\_N\_VWrmsNorm: Create vector of known values, find and validate the weighted root mean square.
- Test\_N\_VWrmsNormMask: Create vector of known values, find and validate the weighted root mean square using all elements except one.
- Test\_N\_VMin: Create vector, find and validate the min.
- Test\_N\_VWL2Norm: Create vector, find and validate the weighted Euclidean L2 norm.
- Test\_N\_VL1Norm: Create vector, find and validate the L1 norm.
- Test\_N\_VCompare: Compare vector with constant returning and validating comparison vector.
- Test\_N\_VInvTest: Test z[i] = 1 / x[i]
- Test\_N\_VConstrMask: Test mask of vector x with vector c.
- Test\_N\_VMinQuotient: Fill two vectors with known values. Calculate and validate minimum quotient.
- Test\_N\_VLinearCombination Case 1a: Test x = a x
- ullet Test\_N\_VLinearCombination Case 1b: Test  $z=a\ x$
- Test\_N\_VLinearCombination Case 2a: Test x = a x + b y
- ullet 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 3c: Test w = a x + b y + c z
- Test\_N\_VScaleAddMulti Case 1a: y = 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 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]
- ullet Test\_N\_VScaleVectorArray Case 1a: z=c
- Test\_N\_VScaleVectorArray Case 1b: Z[i] = c
- Test\_N\_VWrmsNormVectorArray Case 1a: Create a vector of know values, find and validate the weighted root mean square norm.
- Test\_N\_VWrmsNormVectorArray Case 1b: Create a vector array of three vectors of know values, find and validate the weighted root mean square norm of each.
- Test\_N\_VWrmsNormMaskVectorArray Case 1a: Create a vector of know values, find and validate the weighted root mean square norm using all elements except one.
- Test\_N\_VWrmsNormMaskVectorArray Case 1b: Create a vector array of three vectors of know values, find and validate the weighted root mean square norm of each using all elements except one.
- Test\_N\_VScaleAddMultiVectorArray Case 1a: y = a x + y
- Test\_N\_VScaleAddMultiVectorArray Case 1b: z = a x + y
- Test\_N\_VScaleAddMultiVectorArray Case 2a: Y[j][0] = a[j] X[0] + Y[j][0]
- Test\_N\_VScaleAddMultiVectorArray Case 2b: Z[j][0] = a[j] X[0] + Y[j][0]
- Test\_N\_VScaleAddMultiVectorArray Case 3a: Y[0][i] = a[0] X[i] + Y[0][i]
- Test\_N\_VScaleAddMultiVectorArray Case 3b: Z[0][i] = a[0] X[i] + Y[0][i]
- Test\_N\_VScaleAddMultiVectorArray Case 4b: Z[j][i] = a[j] X[i] + Y[j][i]
- ullet Test\_N\_VLinearCombinationVectorArray Case 1a:  $x=a \ x$
- Test\_N\_VLinearCombinationVectorArray Case 1b: z = a x
- ullet 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 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]
- $\bullet \ \, \mathsf{Test\_N\_VLinearCombinationVectorArray} \ \, \mathsf{Case} \ \, 6a: \ \, \mathsf{X}[0][i] = \mathsf{X}[0][i] + \mathsf{c}[1] \ \, \mathsf{X}[1][i] + \mathsf{c}[2] \ \, \mathsf{X}[2][i]$
- Test\_N\_VLinearCombinationVectorArray Case 6b: X[0][i] = c[0] X[0][i] + c[1] X[1][i] + c[2] X[2][i]
- Test\_N\_VLinearCombinationVectorArray Case 6c: Z[i] = c[0] X[0][i] + c[1] X[1][i] + c[2] X[2][i]

- Test\_N\_VDotProdLocal: Calculate MPI task-local portion of the dot product of two vectors.
- Test\_N\_VMaxNormLocal: Create vector with known values, find and validate the MPI task-local portion of the max norm.
- Test\_N\_VMinLocal: Create vector, find and validate the MPI task-local min.
- Test\_N\_VL1NormLocal: Create vector, find and validate the MPI task-local portion of the L1 norm.
- Test\_N\_VWSqrSumLocal: Create vector of known values, find and validate the MPI task-local portion of the weighted squared sum of two vectors.
- Test\_N\_VWSqrSumMaskLocal: Create vector of known values, find and validate the MPI task-local portion of the weighted squared sum of two vectors, using all elements except one.
- Test\_N\_VInvTestLocal: Test the MPI task-local portion of z[i] = 1 / x[i]
- Test\_N\_VConstrMaskLocal: Test the MPI task-local portion of the mask of vector **x** with vector **c**
- Test\_N\_VMinQuotientLocal: Fill two vectors with known values. Calculate and validate the MPI task-local minimum quotient.

# Chapter 8

# Description of the SUNMatrix module

For problems that involve direct methods for solving linear systems, the SUNDIALS solvers not only operate on generic vectors, but also on generic matrices (of type SUNMatrix), through a set of operations defined by the particular SUNMATRIX implementation. Users can provide their own specific implementation of the SUNMATRIX module, particularly in cases where they provide their own NVECTOR and/or linear solver modules, and require matrices that are compatible with those implementations. Alternately, we provide three SUNMATRIX implementations: dense, banded, and sparse. The generic operations are described below, and descriptions of the implementations provided with SUNDIALS follow.

# 8.1 The SUNMatrix API

The SUNMATRIX API can be grouped into two sets of functions: the core matrix operations, and utility functions. Section 8.1.1 lists the core operations, while Section 8.1.2 lists the utility functions.

#### 8.1.1 SUNMatrix core functions

The generic SUNMatrix object defines the following set of core operations:

#### SUNMatGetID

Call id = SUNMatGetID(A);

Description Returns the type identifier for the matrix A. It is used to determine the matrix imple-

mentation type (e.g. dense, banded, sparse,...) from the abstract SUNMatrix interface. This is used to assess compatibility with SUNDIALS-provided linear solver implementa-

tions.

Arguments A (SUNMatrix) a SUNMATRIX object

Return value A SUNMATRIX\_ID, possible values are given in the Table 8.2.

F2003 Name FSUNMatGetID

# SUNMatClone

Call B = SUNMatClone(A);

Description Creates a new SUNMatrix of the same type as an existing matrix A and sets the ops

field. It does not copy the matrix, but rather allocates storage for the new matrix.

Arguments A (SUNMatrix) a SUNMATRIX object

Return value SUNMatrix
F2003 Name FSUNMatClone
F2003 Call type(SUNMatrix), pointer :: B
B => FSUNMatClone(A)

#### SUNMatDestroy

Call SUNMatDestroy(A);

Description Destroys A and frees memory allocated for its internal data.

Arguments A (SUNMatrix) a SUNMATRIX object

Return value None

F2003 Name FSUNMatDestroy

# SUNMatSpace

Call ier = SUNMatSpace(A, &lrw, &liw);

Description Returns the storage requirements for the matrix A. 1rw is a long int containing the

number of realtype words and liw is a long int containing the number of integer words.

Arguments A (SUNMatrix) a SUNMATRIX object

lrw (sunindextype\*) the number of realtype words
liw (sunindextype\*) the number of integer words

Return value None

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

it could be a dummy function in a user-supplied SUNMATRIX module if that information

is not of interest.

F2003 Name FSUNMatSpace

F2003 Call integer(c\_long) :: lrw(1), liw(1)

ier = FSUNMatSpace(A, lrw, liw)

## SUNMatZero

Call ier = SUNMatZero(A);

Description Performs the operation  $A_{ij} = 0$  for all entries of the matrix A.

Arguments A (SUNMatrix) a SUNMATRIX object

Return value A SUNMATRIX return code of type int denoting success/failure

F2003 Name FSUNMatZero

#### SUNMatCopy

Call ier = SUNMatCopy(A,B);

Description Performs the operation  $B_{ij} = A_{i,j}$  for all entries of the matrices A and B.

Arguments A (SUNMatrix) a SUNMATRIX object

B (SUNMatrix) a SUNMATRIX object

Return value A SUNMATRIX return code of type int denoting success/failure

F2003 Name FSUNMatCopy

#### SUNMatScaleAdd

Call ier = SUNMatScaleAdd(c, A, B);

Description Performs the operation A = cA + B.

Arguments c (realtype) constant that scales A

A (SUNMatrix) a SUNMATRIX object

B (SUNMatrix) a SUNMATRIX object

Return value A SUNMATRIX return code of type int denoting success/failure

 $F2003 \; \mathrm{Name} \; \; \mathtt{FSUNMatScaleAdd}$ 

## SUNMatScaleAddI

Call ier = SUNMatScaleAddI(c, A);

Description Performs the operation A = cA + I.

Arguments c (realtype) constant that scales A

A (SUNMatrix) a SUNMATRIX object

Return value A SUNMATRIX return code of type int denoting success/failure

 $F2003 \; Name \; FSUNMatScaleAddI$ 

#### SUNMatMatvecSetup

Call ier = SUNMatMatvecSetup(A);

Description Performs any setup necessary to perform a matrix-vector product. It is useful for

SUNMatrix implementations which need to prepare the matrix itself, or communication

structures before performing the matrix-vector product.

Arguments A (SUNMatrix) a SUNMATRIX object

Return value A SUNMATRIX return code of type int denoting success/failure

F2003 Name FSUNMatMatvecSetup

# SUNMatMatvec

Call ier = SUNMatMatvec(A, x, y);

Description Performs the matrix-vector product operation, y = Ax. It should only be called with

vectors x and y that are compatible with the matrix A - both in storage type and

dimensions.

Arguments A (SUNMatrix) a SUNMATRIX object

x (N\_Vector) a NVECTOR object

y (N\_Vector) an output NVECTOR object

Return value A SUNMATRIX return code of type int denoting success/failure

F2003 Name FSUNMatMatvec

# 8.1.2 SUNMatrix utility functions

To aid in the creation of custom SUNMATRIX modules the generic SUNMATRIX module provides two utility functions SUNMatNewEmpty and SUNMatVCopyOps.

# SUNMatNewEmpty

Call A = SUNMatNewEmpty();

Description The function SUNMatNewEmpty allocates a new generic SUNMATRIX object and initializes

its content pointer and the function pointers in the operations structure to NULL.

Arguments None

Return value This function returns a SUNMatrix object. If an error occurs when allocating the object,

then this routine will return NULL.

F2003 Name FSUNMatNewEmpty

# SUNMatFreeEmpty

Call SUNMatFreeEmpty(A);

Description This routine frees the generic SUNMatrix object, under the assumption that any implementation-

specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the ops pointer is NULL, and, if it is not, it will

free it as well.

Arguments A (SUNMatrix) a SUNMatrix object

Return value None

F2003 Name FSUNMatFreeEmpty

# SUNMatCopyOps

Call retval = SUNMatCopyOps(A, B);

Description The function SUNMatCopyOps copies the function pointers in the ops structure of A into

the ops structure of B.

Arguments A (SUNMatrix) the matrix to copy operations from

B (SUNMatrix) the matrix to copy operations to

Return value This returns 0 if successful and a non-zero value if either of the inputs are NULL or the

ops structure of either input is NULL.

F2003 Name FSUNMatCopyOps

# 8.1.3 SUNMatrix return codes

The functions provided to SUNMATRIX modules within the SUNDIALS-provided SUNMATRIX implementations utilize a common set of return codes, shown in Table 8.1. These adhere to a common pattern: 0 indicates success, and a negative value indicates a failure. The actual values of each return code are primarily to provide additional information to the user in case of a failure.

Table 8.1: Description of the SUNMatrix return codes

Name	Value	Description
SUNMAT_SUCCESS	0	successful call or converged solve
		continued on next page

Matrix ID	Matrix type	ID Value
SUNMATRIX_DENSE	Dense $M \times N$ matrix	0
SUNMATRIX_BAND	Band $M \times M$ matrix	1
SUNMATRIX_MAGMADENSE	Magma dense $\texttt{M} \times \texttt{N}$ matrix	2
SUNMATRIX_SPARSE	Sparse (CSR or CSC) $M \times N$ matrix	3
SUNMATRIX_SLUNRLOC	Adapter for the SuperLU_DIST SuperMatrix	4
SUNMATRIX_CUSPARSE	CUDA sparse CSR matrix	5
SUNMATRIX_CUSTOM	User-provided custom matrix	6

Table 8.2: Identifiers associated with matrix kernels supplied with SUNDIALS.

Name	Value	Description
SUNMAT_ILL_INPUT	-701	an illegal input has been provided to the function
SUNMAT_MEM_FAIL	-702	failed memory access or allocation
SUNMAT_OPERATION_FAIL	-703	a SUNMatrix operation returned nonzero
SUNMAT_MATVEC_SETUP_REQUIRED	-704	the SUNMatMatvecSetup routine needs to be called be-
		fore calling SUNMatMatvec

#### 8.1.4 SUNMatrix identifiers

Each SUNMATRIX implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 8.2. It is recommended that a user-supplied SUNMATRIX implementation use the SUNMATRIX\_CUSTOM identifier.

# 8.1.5 Compatibility of SUNMatrix modules

We note that not all SUNMATRIX types are compatible with all NVECTOR types provided with SUNDIALS. This is primarily due to the need for compatibility within the SUNMatMatvec routine; however, compatibility between SUNMATRIX and NVECTOR implementations is more crucial when considering their interaction within SUNLINSOL objects, as will be described in more detail in Chapter 9. More specifically, in Table 8.3 we show the matrix interfaces available as SUNMATRIX modules, and the compatible vector implementations.

Table 6.9. Severites matrix interfaces and vector implementations that can be used for each								ioi cacii.	
Matrix	Serial	Parallel	OpenMP	pThreads	hypre	PETSC	CUDA	RAJA	User
Interface		(MPI)			Vec.	Vec.			Suppl.
Dense	✓		✓	<b>✓</b>					✓
Band	✓		✓	✓					✓
Sparse	✓		✓	✓					✓
SLUNRloc	✓	$\checkmark$	✓	✓	✓	✓			✓
User supplied	✓	$\checkmark$	✓	✓	✓	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>

Table 8.3: SUNDIALS matrix interfaces and vector implementations that can be used for each

# 8.1.6 The generic SUNMatrix module implementation

The generic SUNMatrix type has been modeled after the object-oriented style of the generic N\_Vector type. Specifically, a generic SUNMatrix is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the matrix, and an *ops* field pointing to a structure with generic matrix operations. The type SUNMatrix is defined as

```
typedef struct _generic_SUNMatrix *SUNMatrix;
struct _generic_SUNMatrix {
    void *content;
    struct _generic_SUNMatrix_Ops *ops;
};
The _generic_SUNMatrix_Ops structure is essentially a list of pointers to the various actual matrix
operations, and is defined as
struct _generic_SUNMatrix_Ops {
  SUNMatrix_ID (*getid)(SUNMatrix);
  SUNMatrix
                (*clone)(SUNMatrix);
  void
                (*destroy)(SUNMatrix);
                (*zero)(SUNMatrix);
  int
                (*copy)(SUNMatrix, SUNMatrix);
  int
                (*scaleadd)(realtype, SUNMatrix, SUNMatrix);
  int
                (*scaleaddi)(realtype, SUNMatrix);
  int
  int
                (*matvecsetup)(SUNMatrix)
  int
                (*matvec)(SUNMatrix, N_Vector, N_Vector);
                (*space)(SUNMatrix, long int*, long int*);
  int
};
```

The generic SUNMATRIX module defines and implements the matrix operations acting on SUNMatrix objects. These routines are nothing but wrappers for the matrix operations defined by a particular SUNMATRIX implementation, which are accessed through the *ops* field of the SUNMatrix structure. To illustrate this point we show below the implementation of a typical matrix operation from the generic SUNMATRIX module, namely SUNMatZero, which sets all values of a matrix A to zero, returning a flag denoting a successful/failed operation:

```
int SUNMatZero(SUNMatrix A)
{
  return((int) A->ops->zero(A));
}
```

Section 8.1.1 contains a complete list of all matrix operations defined by the generic SUNMATRIX module.

The Fortran 2003 interface provides a bind(C) derived-type for the \_generic\_SUNMatrix and the \_generic\_SUNMatrix\_Ops structures. Their definition is given below.

```
type, bind(C), public :: SUNMatrix
type(C_PTR), public :: content
type(C_PTR), public :: ops
end type SUNMatrix
type, bind(C), public :: SUNMatrix_Ops
type(C_FUNPTR), public :: getid
type(C_FUNPTR), public :: clone
type(C_FUNPTR), public :: destroy
type(C_FUNPTR), public :: zero
type(C_FUNPTR), public :: copy
type(C_FUNPTR), public :: scaleadd
type(C_FUNPTR), public :: scaleaddi
type(C_FUNPTR), public :: matvecsetup
type(C_FUNPTR), public :: matvec
type(C_FUNPTR), public :: space
end type SUNMatrix_Ops
```

# 8.1.7 Implementing a custom SUNMatrix

A particular implementation of the Sunmatrix module must:

- Specify the *content* field of the SUNMatrix object.
- Define and implement a minimal subset of the matrix operations. See the documentation for each SUNDIALS solver to determine which SUNMATRIX operations they require.
  - Note that the names of these routines should be unique to that implementation in order to permit using more than one SUNMATRIX module (each with different SUNMatrix internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free a SUNMatrix with the new *content* field and with *ops* pointing to the new matrix operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined SUNMatrix (e.g., a routine to print the content for debugging purposes).
- Optionally, provide accessor macros or functions as needed for that particular implementation to access different parts of the *content* field of the newly defined SUNMatrix.

It is recommended that a user-supplied SUNMATRIX implementation use the SUNMATRIX\_CUSTOM identifier.

To aid in the creation of custom SUNMATRIX modules the generic SUNMATRIX module provides two utility functions SUNMatNewEmpty and SUNMatVCopyOps. When used in custom SUNMATRIX constructors and clone routines these functions will ease the introduction of any new optional matrix operations to the SUNMATRIX API by ensuring only required operations need to be set and all operations are copied when cloning a matrix. These functions are described in Section 8.1.2.

# 8.2 SUNMatrix functions used by IDA

In Table 8.4, we list the matrix functions in the SUNMATRIX module used within the IDA package. The table also shows, for each function, which of the code modules uses the function. The main IDA integrator does not call any SUNMATRIX functions directly, so the table columns are specific to the IDALS interface and the IDABBDPRE preconditioner module. 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 IDA user does not need to know anything about the usage of matrix functions by the IDA code modules in order to use IDA. The information is presented as an implementation detail for the interested reader.

Table 8.4: List of matrix functions usage by IDA code modules

The matrix functions listed in Section 8.1.1 with a † symbol are optionally used, in that these are only called if they are implemented in the SUNMATRIX module that is being used (i.e. their function pointers are non-NULL). The matrix functions listed in Section 8.1.1 that are *not* used by IDA

are: SUNMatCopy, SUNMatClone, SUNMatScaleAdd, SUNMatScaleAddI and SUNMatMatvec. Therefore a user-supplied SUNMATRIX module for IDA could omit these functions.

# 8.3 The SUNMatrix\_Dense implementation

The dense implementation of the SUNMATRIX module provided with SUNDIALS, SUNMATRIX\_DENSE, defines the *content* field of SUNMatrix to be the following structure:

```
struct _SUNMatrixContent_Dense {
  sunindextype M;
  sunindextype N;
  realtype *data;
  sunindextype ldata;
  realtype **cols;
};
These entries of the content field contain the following information:
       - number of rows
       - number of columns
data - pointer to a contiguous block of realtype variables. The elements of the dense matrix are
       stored columnwise, i.e. the (i,j)-th element of a dense sunmatrix A (with 0 \le i < M and 0 \le i < M).
       j < N) may be accessed via data[j*M+i].
ldata - length of the data array (= M \cdot N).
cols - array of pointers. cols[j] points to the first element of the j-th column of the matrix in the
       array data. The (i,j)-th element of a dense sunmatrix A (with 0 \le i < M and 0 \le j < N)
       may be accessed via cols[j][i].
```

The header file to include when using this module is sunmatrix\_dense.h. The SUNMATRIX\_DENSE module is accessible from all SUNDIALS solvers without linking to the libsundials\_sunmatrixdense module library.

# 8.3.1 SUNMatrix\_Dense accessor macros

The following macros are provided to access the content of a SUNMATRIX\_DENSE matrix. The prefix  $SM_{-}$  in the names denotes that these macros are for SUNMatrix implementations, and the suffix  $_{-}D$  denotes that these are specific to the dense version.

#### • SM\_CONTENT\_D

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

The assignment  $A\_cont = SM\_CONTENT\_D(A)$  sets  $A\_cont$  to be a pointer to the dense SUNMatrix content structure.

Implementation:

```
#define SM_CONTENT_D(A) ( (SUNMatrixContent_Dense)(A->content) )
```

SM\_ROWS\_D, SM\_COLUMNS\_D, and SM\_LDATA\_D

These macros give individual access to various lengths relevant to the content of a dense SUNMatrix.

These may be used either to retrieve or to set these values. For example, the assignment A\_rows = SM\_ROWS\_D(A) sets A\_rows to be the number of rows in the matrix A. Similarly, the assignment SM\_COLUMNS\_D(A) = A\_cols sets the number of columns in A to equal A\_cols.

Implementation:

```
#define SM_ROWS_D(A) ( SM_CONTENT_D(A)->M )
```

```
#define SM_COLUMNS_D(A) ( SM_CONTENT_D(A)->N )
#define SM_LDATA_D(A) ( SM_CONTENT_D(A)->ldata )
```

• SM\_DATA\_D and SM\_COLS\_D

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

The assignment A\_data = SM\_DATA\_D(A) sets A\_data to be a pointer to the first component of the data array for the dense SUNMatrix A. The assignment SM\_DATA\_D(A) = A\_data sets the data array of A to be A\_data by storing the pointer A\_data.

Similarly, the assignment  $A\_cols = SM\_COLS\_D(A)$  sets  $A\_cols$  to be a pointer to the array of column pointers for the dense SUNMatrix A. The assignment  $SM\_COLS\_D(A) = A\_cols$  sets the column pointer array of A to be  $A\_cols$  by storing the pointer  $A\_cols$ .

Implementation:

```
#define SM_DATA_D(A) ( SM_CONTENT_D(A)->data )
#define SM_COLS_D(A) ( SM_CONTENT_D(A)->cols )
```

• SM\_COLUMN\_D and SM\_ELEMENT\_D

These macros give access to the individual columns and entries of the data array of a dense SUNMatrix.

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

The assignments SM\_ELEMENT\_D(A,i,j) = a\_ij and a\_ij = SM\_ELEMENT\_D(A,i,j) reference the (i,j)-th element of the M  $\times$  N dense matrix A (with  $0 \le i < M$  and  $0 \le j < N$ ).

Implementation:

```
#define SM_COLUMN_D(A,j) ( (SM_CONTENT_D(A)->cols)[j] )
#define SM_ELEMENT_D(A,i,j) ( (SM_CONTENT_D(A)->cols)[j][i] )
```

# 8.3.2 SUNMatrix\_Dense functions

The SUNMATRIX\_DENSE module defines dense implementations of all matrix operations listed in Section 8.1.1. Their names are obtained from those in Section 8.1.1 by appending the suffix \_Dense (e.g. SUNMatCopy\_Dense). All the standard matrix operations listed in Section 8.1.1 with the suffix \_Dense appended are callable via the FORTRAN 2003 interface by prepending an 'F' (e.g. FSUNMatCopy\_Dense).

The module SUNMATRIX\_DENSE provides the following additional user-callable routines:

#### SUNDenseMatrix

Prototype SUNMatrix SUNDenseMatrix(sunindextype M, sunindextype N)

Description This constructor function creates and allocates memory for a dense SUNMatrix. Its arguments are the number of rows, M, and columns, N, for the dense matrix.

F2003 Name This function is callable as FSUNDenseMatrix when using the Fortran 2003 interface module.

#### SUNDenseMatrix\_Print

Prototype void SUNDenseMatrix\_Print(SUNMatrix A, FILE\* outfile)

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

#### SUNDenseMatrix\_Rows

Prototype sunindextype SUNDenseMatrix\_Rows(SUNMatrix A)

Description This function returns the number of rows in the dense SUNMatrix.

F2003 Name This function is callable as FSUNDenseMatrix\_Rows when using the Fortran 2003 inter-

face module.

## SUNDenseMatrix\_Columns

Prototype sunindextype SUNDenseMatrix\_Columns(SUNMatrix A)

Description This function returns the number of columns in the dense SUNMatrix.

F2003 Name This function is callable as FSUNDenseMatrix\_Columns when using the Fortran 2003

interface module.

#### SUNDenseMatrix\_LData

Prototype sunindextype SUNDenseMatrix\_LData(SUNMatrix A)

Description This function returns the length of the data array for the dense SUNMatrix.

F2003 Name This function is callable as FSUNDenseMatrix\_LData when using the Fortran 2003 inter-

face module.

#### SUNDenseMatrix\_Data

Prototype realtype\* SUNDenseMatrix\_Data(SUNMatrix A)

Description This function returns a pointer to the data array for the dense SUNMatrix.

F2003 Name This function is callable as FSUNDenseMatrix\_Data when using the Fortran 2003 inter-

face module.

# SUNDenseMatrix\_Cols

Prototype realtype\*\* SUNDenseMatrix\_Cols(SUNMatrix A)

Description This function returns a pointer to the cols array for the dense SUNMatrix.

# SUNDenseMatrix\_Column

Prototype realtype\* SUNDenseMatrix\_Column(SUNMatrix A, sunindextype j)

Description This function returns a pointer to the first entry of the jth column of the dense SUNMatrix.

The resulting pointer should be indexed over the range 0 to M-1.

F2003 Name This function is callable as FSUNDenseMatrix\_Column when using the Fortran 2003 in-

terface module.

#### Notes

- When looping over the components of a dense SUNMatrix A, the most efficient approaches are to:
  - First obtain the component array via A\_data = SM\_DATA\_D(A) or A\_data = SUNDenseMatrix\_Data(A) and then access A\_data[i] within the loop.
  - First obtain the array of column pointers via A\_cols = SM\_COLS\_D(A) or A\_cols = SUNDenseMatrix\_Cols(A), and then access A\_cols[j][i] within the loop.
  - Within a loop over the columns, access the column pointer via
     A\_colj = SUNDenseMatrix\_Column(A,j) and then to access the entries within that column using A\_colj[i] within the loop.

All three of these are more efficient than using SM\_ELEMENT\_D(A,i,j) within a double loop.

• Within the SUNMatMatvec\_Dense routine, internal consistency checks are performed to ensure that the matrix is called with consistent NVECTOR implementations. These are currently limited to: NVECTOR\_SERIAL, NVECTOR\_OPENMP, and NVECTOR\_PTHREADS. As additional compatible vector implementations are added to SUNDIALS, these will be included within this compatibility check.



#### 8.3.3 SUNMatrix\_Dense Fortran interfaces

The SUNMATRIX\_DENSE module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

#### FORTRAN 2003 interface module

The fsunmatrix\_dense\_mod FORTRAN module defines interfaces to most SUNMATRIX\_DENSE C functions using the intrinsic iso\_c\_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function SUNDenseMatrix is interfaced as FSUNDenseMatrix.

The Fortran 2003 Sunmatrix\_dense\_mod, and linking to the library libsundials\_fsunmatrix\_dense\_mod.lib in addition to the C library. For details on where the library and module file fsunmatrix\_dense\_mod.mod are installed see Appendix A. We note that the module is accessible from the Fortran 2003 Sundials integrators without separately linking to the libsundials\_fsunmatrixdense\_mod library.

#### FORTRAN 77 interface functions

For solvers that include a Fortran interface module, the Sunmatrix\_dense module also includes the Fortran-callable function FSUNDenseMatInit(code, M, N, ier) to initialize this Sunmatrix\_dense module for a given Sundials solver. Here code is an integer input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); M and N are the corresponding dense matrix construction arguments (declared to match C type long int); and ier is an error return flag equal to 0 for success and -1 for failure. Both code and ier are declared to match C type int. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function FSUNDenseMassMatInit(M, N, ier) initializes this SUNMATRIX\_DENSE module for storing the mass matrix.

# 8.4 The SUNMatrix\_Band implementation

The banded implementation of the SUNMATRIX module provided with SUNDIALS, SUNMATRIX\_BAND, defines the *content* field of SUNMatrix to be the following structure:

```
struct _SUNMatrixContent_Band {
   sunindextype M;
   sunindextype mu;
   sunindextype mu;
   sunindextype ml;
   sunindextype s_mu;
   sunindextype ldim;
   realtype *data;
   sunindextype ldata;
   realtype **cols;
};
```

A diagram of the underlying data representation in a banded matrix is shown in Figure 8.1. A more complete description of the parts of this *content* field is given below:

```
\begin{array}{lll} \texttt{M} & & \text{- number of rows} \\ \texttt{N} & & \text{- number of columns } (\texttt{N} = \texttt{M}) \\ \\ \texttt{mu} & & \text{- upper half-bandwidth, } 0 \leq \texttt{mu} < \texttt{N} \\ \\ \texttt{ml} & & \text{- lower half-bandwidth, } 0 \leq \texttt{ml} < \texttt{N} \end{array}
```

s\_mu - storage upper bandwidth, mu ≤ s\_mu < N. The LU decomposition routines in the associated SUNLINSOL\_BAND and SUNLINSOL\_LAPACKBAND modules write the LU factors into the storage for A. The upper triangular factor U, however, may have an upper bandwidth as big as min(N-1,mu+ml) because of partial pivoting. The s\_mu field holds the upper half-bandwidth allocated for A.

```
ldim - leading dimension (ldim ≥ s_mu+ml+1)
```

data - pointer to a contiguous block of realtype variables. The elements of the banded matrix are stored columnwise (i.e. columns are stored one on top of the other in memory). Only elements within the specified half-bandwidths are stored. data is a pointer to ldata contiguous locations which hold the elements within the band of A.

```
ldata - length of the data array (= ldim \cdot N)
```

cols - array of pointers. cols[j] is a pointer to the uppermost element within the band in the j-th column. This pointer may be treated as an array indexed from s\_mu-mu (to access the uppermost element within the band in the j-th column) to s\_mu+ml (to access the lowest element within the band in the j-th column). Indices from 0 to s\_mu-mu-1 give access to extra storage elements required by the LU decomposition function. Finally, cols[j][i-j+s\_mu] is the (i,j)-th element with  $j-mu \le i \le j+ml$ .

The header file to include when using this module is sunmatrix/sunmatrix\_band.h. The SUNMATRIX\_BAND module is accessible from all SUNDIALS solvers without linking to the libsundials\_sunmatrixband module library.

#### 8.4.1 SUNMatrix Band accessor macros

The following macros are provided to access the content of a SUNMATRIX\_BAND matrix. The prefix SM\_ in the names denotes that these macros are for *SUNMatrix* implementations, and the suffix \_B denotes that these are specific to the *banded* version.

#### • SM\_CONTENT\_B

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

The assignment  $A\_cont = SM\_CONTENT\_B(A)$  sets  $A\_cont$  to be a pointer to the banded SUNMatrix content structure.

Implementation:

```
#define SM_CONTENT_B(A) ((SUNMatrixContent_Band)(A->content) )
```

• SM\_ROWS\_B, SM\_COLUMNS\_B, SM\_UBAND\_B, SM\_LBAND\_B, SM\_SUBAND\_B, SM\_LDIM\_B, and SM\_LDATA\_B These macros give individual access to various lengths relevant to the content of a banded SUNMatrix.

These may be used either to retrieve or to set these values. For example, the assignment A\_rows = SM\_ROWS\_B(A) sets A\_rows to be the number of rows in the matrix A. Similarly, the assignment SM\_COLUMNS\_B(A) = A\_cols sets the number of columns in A to equal A\_cols.

Implementation:

```
#define SM_ROWS_B(A) ( SM_CONTENT_B(A)->M )
#define SM_COLUMNS_B(A) ( SM_CONTENT_B(A)->N )
#define SM_UBAND_B(A) ( SM_CONTENT_B(A)->mu )
#define SM_LBAND_B(A) ( SM_CONTENT_B(A)->ml )
```



Figure 8.1: Diagram of the storage for the SUNMATRIX\_BAND module. Here A is an N  $\times$  N band matrix with upper and lower half-bandwidths 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 linear solver.

```
#define SM_SUBAND_B(A) ( SM_CONTENT_B(A)->s_mu )
#define SM_LDIM_B(A) ( SM_CONTENT_B(A)->ldim )
#define SM_LDATA_B(A) ( SM_CONTENT_B(A)->ldata )
```

SM\_DATA\_B and SM\_COLS\_B

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

The assignment A\_data = SM\_DATA\_B(A) sets A\_data to be a pointer to the first component of the data array for the banded SUNMatrix A. The assignment SM\_DATA\_B(A) = A\_data sets the data array of A to be A\_data by storing the pointer A\_data.

Similarly, the assignment A\_cols = SM\_COLS\_B(A) sets A\_cols to be a pointer to the array of column pointers for the banded SUNMatrix A. The assignment SM\_COLS\_B(A) = A\_cols sets the column pointer array of A to be A\_cols by storing the pointer A\_cols.

Implementation:

```
#define SM_DATA_B(A) ( SM_CONTENT_B(A)->data )
#define SM_COLS_B(A) ( SM_CONTENT_B(A)->cols )
```

• SM\_COLUMN\_B, SM\_COLUMN\_ELEMENT\_B, and SM\_ELEMENT\_B

These macros give access to the individual columns and entries of the data array of a banded SUNMatrix.

The assignments SM\_ELEMENT\_B(A,i,j) = a\_ij and a\_ij = SM\_ELEMENT\_B(A,i,j) reference the (i,j)-th element of the N × N band matrix A, where  $0 \le i, j \le N-1$ . The location (i,j) should further satisfy  $j-mu \le i \le j+ml$ .

The assignment  $col_j = SM\_COLUMN\_B(A,j)$  sets  $col_j$  to be a pointer to the diagonal element of the j-th column of the N × N band matrix A,  $0 \le j \le N-1$ . The type of the expression  $SM\_COLUMN\_B(A,j)$  is realtype \*. The pointer returned by the call  $SM\_COLUMN\_B(A,j)$  can be treated as an array which is indexed from -mu to ml.

The assignments SM\_COLUMN\_ELEMENT\_B(col\_j,i,j) = a\_ij and

a\_ij = SM\_COLUMN\_ELEMENT\_B(col\_j,i,j) reference the (i,j)-th entry of the band matrix A when used in conjunction with SM\_COLUMN\_B to reference the j-th column through col\_j. The index (i,j) should satisfy  $j-mu \le i \le j+ml$ .

Implementation:

# 8.4.2 SUNMatrix\_Band functions

The SUNMATRIX\_BAND module defines banded implementations of all matrix operations listed in Section 8.1.1. Their names are obtained from those in Section 8.1.1 by appending the suffix \_Band (e.g. SUNMatCopy\_Band). All the standard matrix operations listed in Section 8.1.1 with the suffix \_Band appended are callable via the FORTRAN 2003 interface by prepending an 'F' (e.g. FSUNMatCopy\_Band).

The module SUNMATRIX\_BAND provides the following additional user-callable routines:

# SUNBandMatrix

Description This constructor function creates and allocates memory for a banded SUNMatrix. Its

arguments are the matrix size, N, and the upper and lower half-bandwidths of the matrix, mu and ml. The stored upper bandwidth is set to mu+ml to accommodate subsequent factorization in the SUNLINSOL\_BAND and SUNLINSOL\_LAPACKBAND modules.

F2003 Name This function is callable as FSUNBandMatrix when using the Fortran 2003 interface module.

# ${\tt SUNBandMatrixStorage}$

Prototype SUNMatrix SUNBandMatrixStorage(sunindextype N, sunindextype mu, sunindextype ml, sunindextype smu)

Description This constructor function creates and allocates memory for a banded SUNMatrix. Its arguments are the matrix size, N, the upper and lower half-bandwidths of the matrix, mu and ml, and the stored upper bandwidth, smu. When creating a band SUNMatrix, this value should be

- at least min(N-1,mu+ml) if the matrix will be used by the SUNLINSOL\_BAND module;
- exactly equal to mu+ml if the matrix will be used by the SUNLINSOL\_LAPACKBAND module;
- at least mu if used in some other manner.

Note: it is strongly recommended that users call the default constructor, SUNBandMatrix, in all standard use cases. This advanced constructor is used internally within SUNDIALS solvers, and is provided to users who require banded matrices for non-default purposes.

#### SUNBandMatrix\_Print

Prototype void SUNBandMatrix\_Print(SUNMatrix A, FILE\* outfile)

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

## SUNBandMatrix\_Rows

Prototype sunindextype SUNBandMatrix\_Rows(SUNMatrix A)

Description This function returns the number of rows in the banded SUNMatrix.

F2003 Name This function is callable as FSUNBandMatrix\_Rows when using the Fortran 2003 interface module.

# SUNBandMatrix\_Columns

Prototype sunindextype SUNBandMatrix\_Columns(SUNMatrix A)

Description This function returns the number of columns in the banded SUNMatrix.

F2003 Name This function is callable as FSUNBandMatrix\_Columns when using the Fortran 2003 interface module.

## SUNBandMatrix\_LowerBandwidth

Prototype sunindextype SUNBandMatrix\_LowerBandwidth(SUNMatrix A)

Description This function returns the lower half-bandwidth of the banded SUNMatrix.

F2003 Name This function is callable as FSUNBandMatrix\_LowerBandwidth when using the Fortran 2003 interface module.

# SUNBandMatrix\_UpperBandwidth

Prototype sunindextype SUNBandMatrix\_UpperBandwidth(SUNMatrix A)

Description This function returns the upper half-bandwidth of the banded SUNMatrix.

F2003 Name This function is callable as FSUNBandMatrix\_UpperBandwidth when using the Fortran

2003 interface module.

#### SUNBandMatrix\_StoredUpperBandwidth

Prototype sunindextype SUNBandMatrix\_StoredUpperBandwidth(SUNMatrix A)

Description This function returns the stored upper half-bandwidth of the banded SUNMatrix.

F2003 Name This function is callable as FSUNBandMatrix\_StoredUpperBandwidth when using the

Fortran 2003 interface module.

# | SUNBandMatrix\_LDim

Prototype sunindextype SUNBandMatrix\_LDim(SUNMatrix A)

Description This function returns the length of the leading dimension of the banded SUNMatrix.

F2003 Name This function is callable as FSUNBandMatrix\_LDim when using the Fortran 2003 interface

module.

#### SUNBandMatrix\_Data

Prototype realtype\* SUNBandMatrix\_Data(SUNMatrix A)

Description This function returns a pointer to the data array for the banded SUNMatrix.

F2003 Name This function is callable as FSUNBandMatrix\_Data when using the Fortran 2003 interface

module.

# SUNBandMatrix\_Cols

Prototype realtype\*\* SUNBandMatrix\_Cols(SUNMatrix A)

Description This function returns a pointer to the cols array for the banded SUNMatrix.

#### SUNBandMatrix\_Column

Prototype realtype\* SUNBandMatrix\_Column(SUNMatrix A, sunindextype j)

Description This function returns a pointer to the diagonal entry of the j-th column of the banded

 ${\tt SUNMatrix}$ . The resulting pointer should be indexed over the range  $-{\tt mu}$  to  ${\tt ml}$ .

F2003 Name This function is callable as FSUNBandMatrix\_Column when using the Fortran 2003 inter-

face module.

#### Notes

- When looping over the components of a banded SUNMatrix A, the most efficient approaches are to:
  - First obtain the component array via A\_data = SM\_DATA\_B(A) or A\_data = SUNBandMatrix\_Data(A) and then access A\_data[i] within the loop.
  - First obtain the array of column pointers via A\_cols = SM\_COLS\_B(A) or A\_cols = SUNBandMatrix\_Cols(A), and then access A\_cols[j][i] within the loop.
  - Within a loop over the columns, access the column pointer via
     A\_colj = SUNBandMatrix\_Column(A,j) and then to access the entries within that column using SM\_COLUMN\_ELEMENT\_B(A\_colj,i,j).

All three of these are more efficient than using SM\_ELEMENT\_B(A,i,j) within a double loop.

• Within the SUNMatMatvec\_Band routine, internal consistency checks are performed to ensure that the matrix is called with consistent NVECTOR implementations. These are currently limited to: NVECTOR\_SERIAL, NVECTOR\_OPENMP, and NVECTOR\_PTHREADS. As additional compatible vector implementations are added to SUNDIALS, these will be included within this compatibility check.



#### 8.4.3 SUNMatrix Band Fortran interfaces

The SUNMATRIX\_BAND module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

#### FORTRAN 2003 interface module

The fsunmatrix\_band\_mod FORTRAN module defines interfaces to most SUNMATRIX\_BAND C functions using the intrinsic iso\_c\_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function SUNBandMatrix is interfaced as FSUNBandMatrix.

The FORTRAN 2003 SUNMATRIX\_BAND interface module can be accessed with the use statement, i.e. use fsunmatrix\_band\_mod, and linking to the library libsundials\_fsunmatrixband\_mod.lib in addition to the C library. For details on where the library and module file fsunmatrix\_band\_mod.mod are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the libsundials\_fsunmatrixband\_mod library.

#### FORTRAN 77 interface functions

For solvers that include a FORTRAN interface module, the SUNMATRIX\_BAND module also includes the FORTRAN-callable function FSUNBandMatInit(code, N, mu, ml, ier) to initialize this SUNMATRIX\_BAND module for a given SUNDIALS solver. Here code is an integer input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); N, mu, and ml are the corresponding band matrix construction arguments (declared to match C type long int); and ier is an error return flag equal to 0 for success and -1 for failure. Both code and ier are declared to match C type int. Additionally, when using ARKODE with a non-identity mass matrix, the FORTRAN-callable function FSUNBandMassMatInit(N, mu, ml, ier) initializes this SUNMATRIX\_BAND module for storing the mass matrix.

# 8.5 The SUNMatrix\_Sparse implementation

The sparse implementation of the SUNMATRIX module provided with SUNDIALS, SUNMATRIX\_SPARSE, is designed to work with either *compressed-sparse-column* (CSC) or *compressed-sparse-row* (CSR) sparse matrix formats. To this end, it defines the *content* field of SUNMatrix to be the following structure:

```
struct _SUNMatrixContent_Sparse {
   sunindextype M;
   sunindextype N;
   sunindextype NNZ;
   sunindextype NP;
   realtype *data;
   int sparsetype;
   sunindextype *indexvals;
   sunindextype *indexptrs;
   /* CSC indices */
   sunindextype **rowvals;
```

```
sunindextype **colptrs;
/* CSR indices */
sunindextype **colvals;
sunindextype **rowptrs;
};
```

A diagram of the underlying data representation for a CSC matrix is shown in Figure 8.2 (the CSR format is similar). A more complete description of the parts of this *content* field is given below:

M - number of rowsN - number of columns

NNZ - maximum number of nonzero entries in the matrix (allocated length of data and indexvals arrays)

NP - number of index pointers (e.g. number of column pointers for CSC matrix). For CSC matrices NP = N, and for CSR matrices NP = M. This value is set automatically based the input for sparsetype.

- 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)

- 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

- pointer to a contiguous block of int variables (of length NP+1). For CSC matrices each entry provides the index of the first column entry into the data and indexvals arrays, e.g. if indexptr[3]=7, then the first nonzero entry in the fourth column of the matrix is located in data[7], and is located in row indexvals[7] of the matrix. The last entry contains the total number of nonzero values in the matrix and hence points one past the end of the active data in the data and indexvals arrays. For CSR matrices, each entry provides the index of the first row entry into the data and indexvals arrays.

The following pointers are added to the SlsMat type for user convenience, to provide a more intuitive interface to the CSC and CSR sparse matrix data structures. They are set automatically when creating a sparse SUNMATRIX, based on the sparse matrix storage type.

rowvals - pointer to indexvals when sparsetype is CSC\_MAT, otherwise set to NULL.

colptrs - pointer to indexptrs when sparsetype is CSC\_MAT, otherwise set to NULL.

colvals - pointer to indexvals when sparsetype is CSR\_MAT, otherwise set to NULL.

rowptrs - pointer to indexptrs when sparsetype is CSR\_MAT, otherwise set to NULL. For example, the  $5\times4$  CSC matrix

 $\begin{bmatrix}
0 & 3 & 1 & 0 \\
3 & 0 & 0 & 2 \\
0 & 7 & 0 & 0 \\
1 & 0 & 0 & 9 \\
0 & 0 & 0 & 5
\end{bmatrix}$ 

could be stored in this structure as either

```
M = 5;
N = 4;
NNZ = 8;
NP = N;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0};
sparsetype = CSC_MAT;
indexvals = {1, 3, 0, 2, 0, 1, 3, 4};
indexptrs = {0, 2, 4, 5, 8};
```

```
M = 5;
N = 4;
NNZ = 10;
NP = N;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0, *, *};
sparsetype = CSC_MAT;
indexvals = {1, 3, 0, 2, 0, 1, 3, 4, *, *};
indexptrs = {0, 2, 4, 5, 8};
```

where the first has no unused space, and the second has additional storage (the entries marked with \* may contain any values). Note in both cases that the final value in indexptrs is 8, indicating the total number of nonzero entries in the matrix.

Similarly, in CSR format, the same matrix could be stored as

```
M = 5;
N = 4;
NNZ = 8;
NP = M;
data = {3.0, 1.0, 3.0, 2.0, 7.0, 1.0, 9.0, 5.0};
sparsetype = CSR_MAT;
indexvals = {1, 2, 0, 3, 1, 0, 3, 3};
indexptrs = {0, 2, 4, 5, 7, 8};
```

The header file to include when using this module is sunmatrix/sunmatrix\_sparse.h. The SUNMATRIX\_SPARSE module is accessible from all SUNDIALS solvers without linking to the libsundials\_sunmatrixsparse module library.

# 8.5.1 SUNMatrix\_Sparse accessor macros

The following macros are provided to access the content of a SUNMATRIX\_SPARSE matrix. The prefix SM\_ in the names denotes that these macros are for *SUNMatrix* implementations, and the suffix \_S denotes that these are specific to the *sparse* version.

#### • SM\_CONTENT\_S

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

The assignment  $A\_cont = SM\_CONTENT\_S(A)$  sets  $A\_cont$  to be a pointer to the sparse SUNMatrix content structure.

Implementation:

```
#define SM_CONTENT_S(A) ((SUNMatrixContent_Sparse)(A->content) )
```

• SM\_ROWS\_S, SM\_COLUMNS\_S, SM\_NNZ\_S, SM\_NP\_S, and SM\_SPARSETYPE\_S

These macros give individual access to various lengths relevant to the content of a sparse SUNMatrix.

These may be used either to retrieve or to set these values. For example, the assignment A\_rows = SM\_ROWS\_S(A) sets A\_rows to be the number of rows in the matrix A. Similarly, the assignment SM\_COLUMNS\_S(A) = A\_cols sets the number of columns in A to equal A\_cols.

Implementation:

```
#define SM_ROWS_S(A) ( SM_CONTENT_S(A)->M )
#define SM_COLUMNS_S(A) ( SM_CONTENT_S(A)->N )
#define SM_NNZ_S(A) ( SM_CONTENT_S(A)->NNZ )
#define SM_NP_S(A) ( SM_CONTENT_S(A)->NP )
#define SM_SPARSETYPE_S(A) ( SM_CONTENT_S(A)->sparsetype )
```



Figure 8.2: Diagram of the storage for a compressed-sparse-column matrix. Here A is an  $M \times N$  sparse matrix with storage for up to NNZ nonzero entries (the allocated length of both data and indexvals). The entries in indexvals may assume values from 0 to M-1, corresponding to the row index (zero-based) of each nonzero value. The entries in data contain the values of the nonzero entries, with the row i, column j entry of A (again, zero-based) denoted as A(i,j). The indexptrs array contains N+1 entries; the first N denote the starting index of each column within the indexvals and data arrays, while the final entry points one past the final nonzero entry. Here, although NNZ values are allocated, only nz are actually filled in; the greyed-out portions of data and indexvals indicate extra allocated space.

#### • SM\_DATA\_S, SM\_INDEXVALS\_S, and SM\_INDEXPTRS\_S

These macros give access to the data and index arrays for the matrix entries.

The assignment A\_data = SM\_DATA\_S(A) sets A\_data to be a pointer to the first component of the data array for the sparse SUNMatrix A. The assignment SM\_DATA\_S(A) = A\_data sets the data array of A to be A\_data by storing the pointer A\_data.

Similarly, the assignment A\_indexvals = SM\_INDEXVALS\_S(A) sets A\_indexvals to be a pointer to the array of index values (i.e. row indices for a CSC matrix, or column indices for a CSR matrix) for the sparse SUNMatrix A. The assignment A\_indexptrs = SM\_INDEXPTRS\_S(A) sets A\_indexptrs to be a pointer to the array of index pointers (i.e. the starting indices in the data/indexvals arrays for each row or column in CSR or CSC formats, respectively).

Implementation:

```
#define SM_DATA_S(A) ( SM_CONTENT_S(A)->data )
#define SM_INDEXVALS_S(A) ( SM_CONTENT_S(A)->indexvals )
#define SM_INDEXPTRS_S(A) ( SM_CONTENT_S(A)->indexptrs )
```

# 8.5.2 SUNMatrix\_Sparse functions

The SUNMATRIX\_SPARSE module defines sparse implementations of all matrix operations listed in Section 8.1.1. Their names are obtained from those in Section 8.1.1 by appending the suffix \_Sparse (e.g. SUNMatCopy\_Sparse). All the standard matrix operations listed in Section 8.1.1 with the suffix \_Sparse appended are callable via the FORTRAN 2003 interface by prepending an 'F' (e.g. FSUNMatCopy\_Sparse).

The module SUNMATRIX\_SPARSE provides the following additional user-callable routines:

#### SUNSparseMatrix

```
Prototype SUNMatrix SUNSparseMatrix(sunindextype M, sunindextype N, sunindextype NNZ, int sparsetype)
```

Description This function creates and allocates memory for a sparse SUNMatrix. Its arguments are the number of rows and columns of the matrix, M and N, the maximum number of nonzeros to be stored in the matrix, NNZ, and a flag sparsetype indicating whether to use CSR or CSC format (valid arguments are CSR\_MAT or CSC\_MAT).

F2003 Name This function is callable as FSUNSparseMatrix when using the Fortran 2003 interface module.

# ${\tt SUNSparseFromDenseMatrix}$

```
Prototype SUNMatrix SUNSparseFromDenseMatrix(SUNMatrix A, realtype droptol, int sparsetype);
```

Description This function creates a new sparse matrix from an existing dense matrix by copying all values with magnitude larger than droptol into the sparse matrix structure.

Requirements:

- A must have type SUNMATRIX\_DENSE;
- droptol must be non-negative;
- sparsetype must be either CSC\_MAT or CSR\_MAT.

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

F2003 Name This function is callable as FSUNSparseFromDenseMatrix when using the Fortran 2003 interface module.

#### SUNSparseFromBandMatrix

Prototype SUNMatrix SUNSparseFromBandMatrix(SUNMatrix A, realtype droptol, int sparsetype);

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

Requirements:

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

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

F2003 Name This function is callable as FSUNSparseFromBandMatrix when using the Fortran 2003 interface module.

#### SUNSparseMatrix\_Realloc

Prototype int SUNSparseMatrix\_Realloc(SUNMatrix A)

Description This function reallocates internal storage arrays in a sparse matrix so that the resulting sparse matrix has no wasted space (i.e. the space allocated for nonzero entries equals the actual number of nonzeros, indexptrs[NP]). Returns 0 on success and 1 on failure (e.g. if the input matrix is not sparse).

F2003 Name This function is callable as FSUNSparseMatrix\_Realloc when using the Fortran 2003 interface module.

# SUNSparseMatrix\_Reallocate

Prototype int SUNSparseMatrix\_Reallocate(SUNMatrix A, sunindextype NNZ)

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

F2003 Name This function is callable as FSUNSparseMatrix\_Reallocate when using the Fortran 2003 interface module.

# SUNSparseMatrix\_Print

Prototype void SUNSparseMatrix\_Print(SUNMatrix A, FILE\* outfile)

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

#### SUNSparseMatrix\_Rows

Prototype sunindextype SUNSparseMatrix\_Rows(SUNMatrix A)

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

F2003 Name This function is callable as FSUNSparseMatrix\_Rows when using the Fortran 2003 interface module.

# SUNSparseMatrix\_Columns

Prototype sunindextype SUNSparseMatrix\_Columns(SUNMatrix A)

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

F2003 Name This function is callable as FSUNSparseMatrix\_Columns when using the Fortran 2003

interface module.

# SUNSparseMatrix\_NNZ

Prototype sunindextype SUNSparseMatrix\_NNZ(SUNMatrix A)

Description This function returns the number of entries allocated for nonzero storage for the sparse

matrix SUNMatrix.

F2003 Name This function is callable as FSUNSparseMatrix\_NNZ when using the Fortran 2003 inter-

face module.

#### SUNSparseMatrix\_NP

Prototype sunindextype SUNSparseMatrix\_NP(SUNMatrix A)

Description This function returns the number of columns/rows for the sparse SUNMatrix, depending

on whether the matrix uses CSC/CSR format, respectively. The indexptrs array has

NP+1 entries.

F2003 Name This function is callable as FSUNSparseMatrix\_NP when using the Fortran 2003 interface

module.

# ${\tt SUNSparseMatrix\_SparseType}$

Prototype int SUNSparseMatrix\_SparseType(SUNMatrix A)

Description This function returns the storage type (CSR\_MAT or CSC\_MAT) for the sparse SUNMatrix.

F2003 Name This function is callable as FSUNSparseMatrix\_SparseType when using the Fortran 2003

interface module.

# SUNSparseMatrix\_Data

Prototype realtype\* SUNSparseMatrix\_Data(SUNMatrix A)

Description This function returns a pointer to the data array for the sparse SUNMatrix.

F2003 Name This function is callable as FSUNSparseMatrix\_Data when using the Fortran 2003 inter-

face module.

#### SUNSparseMatrix\_IndexValues

Prototype sunindextype\* SUNSparseMatrix\_IndexValues(SUNMatrix A)

Description This function returns a pointer to index value array for the sparse SUNMatrix: for CSR

format this is the column index for each nonzero entry, for CSC format this is the row

index for each nonzero entry.

F2003 Name This function is callable as FSUNSparseMatrix\_IndexValues when using the Fortran

2003 interface module.

# SUNSparseMatrix\_IndexPointers

Prototype sunindextype\* SUNSparseMatrix\_IndexPointers(SUNMatrix A)

Description This function returns a pointer to the index pointer array for the sparse SUNMatrix:

for CSR format this is the location of the first entry of each row in the data and indexvalues arrays, for CSC format this is the location of the first entry of each column.

F2003 Name This function is callable as FSUNSparseMatrix\_IndexPointers when using the Fortran

2003 interface module.

Within the SUNMatMatvec\_Sparse routine, internal consistency checks are performed to ensure that the matrix is called with consistent NVECTOR implementations. These are currently limited to: NVECTOR\_SERIAL, NVECTOR\_OPENMP, NVECTOR\_PTHREADS, and NVECTOR\_CUDA when using managed memory. As additional compatible vector implementations are added to SUNDIALS, these will be included within this compatibility check.

# 8.5.3 SUNMatrix\_Sparse Fortran interfaces

The SUNMATRIX\_SPARSE module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

#### FORTRAN 2003 interface module

The fsunmatrix\_sparse\_mod FORTRAN module defines interfaces to most SUNMATRIX\_SPARSE C functions using the intrinsic iso\_c\_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function SUNSparseMatrix is interfaced as FSUNSparseMatrix.

The Fortran 2003 Sunmatrix\_sparse interface module can be accessed with the use statement, i.e. use fsunmatrix\_sparse\_mod, and linking to the library libsundials\_fsunmatrixsparse\_mod.lib in addition to the C library. For details on where the library and module file fsunmatrix\_sparse\_mod.mod are installed see Appendix A. We note that the module is accessible from the Fortran 2003 Sundials integrators without separately linking to the libsundials\_fsunmatrixsparse\_mod library.

## FORTRAN 77 interface functions

For solvers that include a Fortran interface module, the SUNMATRIX\_SPARSE module also includes the Fortran-callable function FSUNSparseMatInit(code, M, N, NNZ, sparsetype, ier) to initialize this SUNMATRIX\_SPARSE module for a given SUNDIALS solver. Here code is an integer input for the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); M, N and NNZ are the corresponding sparse matrix construction arguments (declared to match C type long int); sparsetype is an integer flag indicating the sparse storage type (0 for CSC, 1 for CSR); and ier is an error return flag equal to 0 for success and -1 for failure. Each of code, sparsetype and ier are declared so as to match C type int. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function FSUNSparseMassMatInit(M, N, NNZ, sparsetype, ier) initializes this SUNMATRIX\_SPARSE module for storing the mass matrix.

# 8.6 The SUNMatrix\_SLUNRloc implementation

The SUNMATRIX\_SLUNRLOC implementation of the SUNMATRIX module provided with SUNDIALS is an adapter for the SuperMatrix structure provided by the SuperLU\_DIST sparse matrix factorization and solver library written by X. Sherry Li [8, 28, 40, 41]. It is designed to be used with the SUNLINSOL\_SUPERLUDIST linear solver discussed in Section 9.10. To this end, it defines the *content* field of SUNMatrix to be the following structure:



A more complete description of the this *content* field is given below:

own\_data - a flag which indicates if the SUNMatrix is responsible for freeing A\_super

grid - pointer to the SuperLU\_DIST structure that stores the 2D process grid

row\_to\_proc - a mapping between the rows in the matrix and the process it resides on; will be NULL
until the SUNMatMatvecSetup routine is called

gsmv\_comm - pointer to the SuperLU\_DIST structure that stores the communication information
needed for matrix-vector multiplication; will be NULL until the SUNMatMatvecSetup routine is
called

A\_super - pointer to the underlying SuperLU\_DIST SuperMatrix with Stype = SLU\_NR\_loc, Dtype = SLU\_D, Mtype = SLU\_GE; must have the full diagonal present to be used with SUNMatScaleAddI routine

ACS\_super - a column-sorted version of the matrix needed to perform matrix-vector multiplication; will be NULL until the routine SUNMatMatvecSetup routine is called

The header file to include when using this module is sunmatrix\_sunmatrix\_slunrloc.h. The installed module library to link to is libsundials\_sunmatrixslunrloc.lib where .lib is typically .so for shared libraries and .a for static libraries.

# 8.6.1 SUNMatrix\_SLUNRloc functions

The module SUNMATRIX\_SLUNRLOC provides the following user-callable routines:

#### SUNMatrix\_SLUNRloc

Call A = SUNMatrix\_SLUNRloc(Asuper, grid);

 $\label{locates} \textbf{Description} \quad \text{The function $\tt SUNMatrix\_SLUNRloc} \ creates \ and \ allocates \ memory \ for \ a \ SUNMATRIX\_SLUNRLOC$ 

object.

Arguments Asuper (SuperMatrix\*) a fully-allocated SuperLU\_DIST SuperMatrix that the SUN-

Matrix will wrap; must have Stype = SLU\_NR\_loc, Dtype = SLU\_D, Mtype = SLU\_GE

to be compatible

grid (gridinfo\_t\*) the initialized SuperLU\_DIST 2D process grid structure

Return value a SUNMatrix object if Asuper is compatible else NULL

Notes

#### SUNMatrix\_SLUNRloc\_Print

Call SUNMatrix\_SLUNRloc\_Print(A, fp);

Description The function SUNMatrix\_SLUNRloc\_Print prints the underlying SuperMatrix content.

Arguments A (SUNMatrix) the matrix to print

fp (FILE) the file pointer used for printing

Return value void

Notes

# SUNMatrix\_SLUNRloc\_SuperMatrix

Call Asuper = SUNMatrix\_SLUNRloc\_SuperMatrix(A);

Description The function SUNMatrix\_SLUNRloc\_SuperMatrix provides access to the underlying Su-

perLU\_DIST SuperMatrix of A.

Arguments A (SUNMatrix) the matrix to access

Return value SuperMatrix\*

Notes

# SUNMatrix\_SLUNRloc\_ProcessGrid

Call grid = SUNMatrix\_SLUNRloc\_ProcessGrid(A);

Description The function SUNMatrix\_SLUNRloc\_ProcessGrid provides access to the SuperLU\_DIST

gridinfo\_t structure associated with A.

Arguments A (SUNMatrix) the matrix to access

Return value gridinfo\_t\*

Notes

#### SUNMatrix\_SLUNRloc\_OwnData

Call does\_own\_data = SUNMatrix\_SLUNRloc\_OwnData(A);

Description The function SUNMatrix\_SLUNRloc\_OwnData returns true if the SUNMatrix object is

responsible for freeing A\_super, otherwise it returns false.

Arguments A (SUNMatrix) the matrix to access

Return value booleantype

Notes

The SUNMATRIX\_SLUNRLOC module defines implementations of all generic SUNMatrix operations listed in Section 8.1.1:

- SUNMatGetID\_SLUNRloc returns SUNMATRIX\_SLUNRLOC
- SUNMatClone\_SLUNRloc
- SUNMatDestroy\_SLUNRloc
- SUNMatSpace\_SLUNRloc this only returns information for the storage within the matrix interface, i.e. storage for row\_to\_proc
- SUNMatZero\_SLUNRloc
- SUNMatCopy\_SLUNRloc
- SUNMatScaleAdd\_SLUNRloc performs A = cA + B, but A and B must have the same sparsity pattern
- SUNMatScaleAddI\_SLUNRloc performs A = cA + I, but the diagonal of A must be present
- SUNMatMatvecSetup\_SLUNRloc initializes the SuperLU\_DIST parallel communication structures needed to perform a matrix-vector product; only needs to be called before the first call to SUNMatMatvec or if the matrix changed since the last setup
- SUNMatMatvec\_SLUNRloc



The SUNMATRIX\_SLUNRLOC module requires that the complete diagonal, i.e. nonzeros and zeros, is present in order to use the SUNMatScaleAddI operation.

# 8.7 The SUNMatrix\_cuSparse implementation

The SUNMATRIX\_CUSPARSE implementation of the SUNMatrix module provided with SUNDIALS, is an interface to the NVIDIA cusparse matrix for use on NVIDIA GPUs [7]. All data stored by this matrix implementation resides on the GPU at all times. The implementation currently supports the cusparse CSR matrix format described in the cusparse documentation as well as a unique low-storage format for block-diagonal matrices of the form

$$\mathbf{A} = \begin{bmatrix} \mathbf{A_0} & 0 & \cdots & 0 \\ 0 & \mathbf{A_1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A_{n-1}} \end{bmatrix}$$

where all the block matrices  $A_j$  share the same sparsity pattern. We will refer to this format as BCSR (not to be confused with the canonical BSR format where each block is stored as dense). In this format, the CSR column indices and row pointers are only stored for the first block and are computed only as necessary for other blocks. This can drastically reduce the amount of storage required compared to the regular CSR format when there is a large number of blocks. This format is well-suited for, and intended to be used with the SUNLinearSolver\_cuSolverSp\_batchQR linear solver (see Section 9.12).

The header file to include when using this module is sunmatrix/sunmatrix\_cusparse.h. The installed library to link to is libsundials\_sunmatrixcusparse.lib where .lib is typically .so for shared libraries and .a for static libraries.

The SUNMatrix\_cuSparse module is experimental and subject to change.



# 8.7.1 SUNMatrix\_cuSparse functions

The SUNMATRIX\_CUSPARSE module defines GPU-enabled sparse implementations of all matrix operations listed in Section 8.1.1 except for the SUNMatSpace and SUNMatMatvecSetup operations:

- 1. SUNMatGetID\_cuSparse returns SUNMATRIX\_CUSPARSE
- 2. SUNMatClone\_cuSparse
- 3. SUNMatDestroy\_cuSparse
- 4. SUNMatZero\_cuSparse
- 5. SUNMatCopy\_cuSparse
- 6. SUNMatScaleAdd\_cuSparse performs A = cA + B, where A and B must have the same sparsity pattern
- 7. SUNMatScaleAddI\_cuSparse performs A = cA + I, where the diagonal of A must be present
- 8. SUNMatMatvec\_cuSparse

In addition, the SUNMATRIX\_CUSPARSE module defines the following implementation specific functions:

# SUNMatrix\_cuSparse\_NewCSR

Call A = SUNMatrix\_cuSparse\_NewCSR(M, N, NNZ, cusp)

Description This constructor function creates and allocates memory for a SUNMATRIX\_CUSPARSE SUNMatrix that uses the CSR storage format.

Arguments M (int) the number of matrix rows

N (int) the number of matrix columns

NNZ (int) the number of matrix nonzeros cusp (cusparseHandle\_t) a valid cusparseHandle\_t

Return value a SUNMatrix object if successful else NULL

#### SUNMatrix\_cuSparse\_NewBlockCSR

Call A = SUNMatrix\_cuSparse\_NewBlockCSR(nblocks, blockrows, blockcols, blocknnz,

cusp)

Description This constructor function creates and allocates memory for a SUNMATRIX\_CUSPARSE

SUNMatrix that leverages the SUNMAT\_CUSPARSE\_BCSR storage format to store a block diagonal matrix where each block shares the same sparsity pattern. The blocks must

be square.

Arguments nblocks (int) the number of matrix blocks

blockrows (int) the number of rows for a block blockcols (int) the number of columns for a block blocknnz (int) the number of nonzeros in a block

cusp a valid cusparseHandle\_t

Return value a SUNMatrix object if successful else NULL

Notes The SUNMAT\_CUSPARSE\_BCSR format currently only supports square matrices.

# SUNMatrix\_cuSparse\_MakeCSR

Call A = SUNMatrix\_cuSparse\_MakeCSR(mat\_descr, M, N, NNZ, rowptrs, colind, data,

cusp)

Description This constructor function creates and allocates memory for a SUNMATRIX\_CUSPARSE

SUNMatrix that uses the CSR storage format from the user provided pointers.

Arguments mat\_decsr a valid cusparseMatDescr\_t object; must use CUSPARSE\_INDEX\_BASE\_ZERO

indexing

M (int) the number of matrix rows
N (int) the number of matrix columns
NNZ (int) the number of matrix nonzeros

rowptrs (int\*)a contiguous array of the CSR row pointers
colind (int\*) a contiguous array of the CSR column indices
data (realtype\*) a contiguous array of the nonzero data
cusp (cusparseHandle\_t) a valid cusparseHandle\_t

Return value a SUNMatrix object if successful else NULL

# SUNMatrix\_cuSparse\_Rows

Call M = SUNMatrix\_cuSparse\_Rows(A)

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

Arguments A (SUNMatrix)

Return value the number of rows in the sparse SUNMatrix

#### SUNMatrix\_cuSparse\_Columns

Call N = SUNMatrix\_cuSparse\_Columns(A)

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

Arguments A (SUNMatrix)

Return value the number of columns in the sparse SUNMatrix

# SUNMatrix\_cuSparse\_NNZ

Call nnz = SUNMatrix\_cuSparse\_NNZ(A)

Description This function returns the number of nonzeros in the sparse SUNMatrix.

Arguments A (SUNMatrix)

Return value the number of nonzeros in the sparse SUNMatrix

# ${\tt SUNMatrix\_cuSparse\_SparseType}$

Call type = SUNMatrix\_cuSparse\_SparseType(A)

Description This function returns the sparsity format for the sparse SUNMatrix.

Arguments A (SUNMatrix)

Return value the SUNMAT\_CUSPARSE\_CSR or SUNMAT\_CUSPARSE\_BCSR sparsity formats

# SUNMatrix\_cuSparse\_IndexValues

Call colind = SUNMatrix\_cuSparse\_IndexValues(A)

Description This function returns a pointer to the index value array for the sparse SUNMatrix.

Arguments A (SUNMatrix)

Return value for the CSR format this is an array of the column indices for each nonzero entry. For

the BCSR format this is an array of the column indices for each nonzero entry in the

first block only.

# ${\tt SUNMatrix\_cuSparse\_IndexPointers}$

Call rowptrs = SUNMatrix\_cuSparse\_IndexPointers(A)

Description This function returns a pointer to the index pointers array for the sparse SUNMatrix.

Arguments A (SUNMatrix)

Return value for the CSR format this is an array of the locations of the first entry of each row in the

data and indexvalues arrays, for the BCSR format this is an array of the locations of

each row in the data and indexvalues arrays in the first block only.

# SUNMatrix\_cuSparse\_NumBlocks

Call nblocks = SUNMatrix\_cuSparse\_NumBlocks(A)

Description This function returns the number of blocks in the sparse SUNMatrix.

Arguments A (SUNMatrix)

Return value the number of matrix blocks

#### |SUNMatrix\_cuSparse\_BlockRows

Call blockrows = SUNMatrix\_cuSparse\_BlockRows(A)

Description This function returns the number of rows of a block of the sparse SUNMatrix.

Arguments A (SUNMatrix)

Return value the number of rows of a block

#### SUNMatrix\_cuSparse\_BlockColumns

Call blockrows = SUNMatrix\_cuSparse\_BlockColumns(A)

Description This function returns the number of columns of a block of the sparse SUNMatrix.

Arguments A (SUNMatrix)

Return value the number of columns of a block

#### SUNMatrix\_cuSparse\_BlockNNZ

Call blockdim = SUNMatrix\_cuSparse\_BlockNNZ(A)

Description This function returns the nonzeros of a block of the sparse SUNMatrix.

Arguments A (SUNMatrix)

Return value the number of nonzeros of a block

#### SUNMatrix\_cuSparse\_BlockData

Call nzdata = SUNMatrix\_cuSparse\_BlockData(A, blockidx)

Description This function returns a pointer to the start of the nonzero values in the data array for

given block index. The first block in the SUNMatrix is index 0, the second block is index

1, and so on.

Arguments A (SUNMatrix)

blockidx (int) the index of the desired block

Return value a pointer to the start of the nonzero values in the data array for given block index

#### SUNMatrix\_cuSparse\_CopyToDevice

Call retval = SUNMatrix\_cuSparse\_CopyToDevice(A, h\_data, h\_idxptrs, h\_idxvals)

Description This 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.

Arguments A (SUNMatrix)

h\_data (realtype\*) a pointer to an allocated array of at least SUNMatrix\_cuSparse\_NNZ(A)

 $\boldsymbol{*}$   $\boldsymbol{\mathtt{sizeof}}$  (realtype) bytes; the nonzero values will be copied from this array

onto the device

h\_idxptrs (int\*) a pointer to an allocated array of at least (SUNMatrix\_cuSparse\_BlockDim(A)+1)

\* sizeof(int) bytes; the index pointers will be copied from this array onto

the device

h\_idxvals (int\*) a pointer to an allocated array of at least SUNMatrix\_cuSparse\_BlockNNZ(A)

\* sizeof(int) bytes; the index values will be copied from this array onto

the device

Return value SUNMAT\_SUCCESS if the copy operation(s) were successful, or a nonzero error code otherwise.

#### SUNMatrix\_cuSparse\_CopyFromDevice

Call retval = SUNMatrix\_cuSparse\_CopyFromDevice(A, h\_data, h\_idxptrs, h\_idxvals)

Description This 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.

Arguments A (SUNMatrix)

h\_data (realtype\*) a pointer to an allocated array of at least SUNMatrix\_cuSparse\_NNZ(A)

\* sizeof(realtype) bytes; the nonzero values will be copied into this array

from the device

h\_idxptrs (int\*) a pointer to an allocated array of at least (SUNMatrix\_cuSparse\_BlockDim(A)+1)

\* sizeof(int) bytes; the index pointers will be copied into this array from

the device

h\_idxvals (int\*) a pointer to an allocated array of at least SUNMatrix\_cuSparse\_BlockNNZ(A)

 $\boldsymbol{\ast}$  sizeof(int) bytes; the index values will be copied into this array from

the device

Return value SUNMAT\_SUCCESS if the copy operation(s) were successful, or a nonzero error code otherwise.

#### SUNMatrix\_cuSparse\_SetKernelExecPolicy

Call retval = SUNMatrix\_cuSparse\_SetKernelExecPolicy(A, exec\_policy);

Description This function sets the execution policies which control the kernel parameters utilized

when launching the CUDA kernels. By default the matrix is setup to use a policy which tries to leverage the structure of the matrix. See section 7.9.2 for more information

about the  ${\tt SUNCudaExecPolicy}$  class.

Arguments A (SUNMatrix)

exec\_policy (SUNCudaExecPolicy\*)

Return value SUNMAT\_SUCCESS if the operation(s) were successful, or a nonzero error code otherwise.

Notes All matrices and vector used in a single instance of a SUNDIALS solver must use the same CUDA stream, and the CUDA stream must be set prior to solver initialization.

#### SUNMatrix\_cuSparse\_SetFixedPattern

Call retval = SUNMatrix\_cuSparse\_SetFixedPattern(A, yesno)

Description This function changes the behavior of the the SUNMatZero operation on the SUNMatrix

object A. By default the matrix sparsity pattern is not considered to be fixed, thus, the SUNMatZero operation zeros out all data array as well as the indexvalues and indexpointers arrays. Providing a value of 1 or SUNTRUE for the yesno argument changes the behavior of SUNMatZero on A so that only the data is zeroed out, but not the indexvalues or indexpointers arrays. Providing a value of 0 or SUNFALSE for the

yesno argument is equivalent to the default behavior.

Arguments A (SUNMatrix)

yesno (booleantype)

Return value SUNMAT\_SUCCESS if the operation(s) were successful, or a nonzero error code otherwise.

# 8.7.2 SUNMatrix\_cuSparse Usage Notes

The SUNMATRIX\_CUSPARSE module only supports 32-bit indexing, thus SUNDIALS must be built for 32-bit indexing to use this module.

The SUNMATRIX\_CUSPARSE module can be used with CUDA streams by calling the cuSPARSE function cusparseSetStream on the the cusparseHandle\_t that is provided to the SUNMATRIX\_CUSPARSE constructor.

When using the SUNMATRIX\_CUSPARSE module with a SUNDIALS package (e.g. CVODE), the stream given to cuSPARSE should be the same stream used for the NVECTOR object that is provided to the package, and the NVECTOR object given to the SUNMatvec operation. If different streams are utilized, synchronization issues may occur.



# 8.8 The SUNMATRIX\_MAGMADENSE implementation

The SUNMATRIX\_MAGMADENSE implementation of the SUNDIALS SUNMatrix API interfaces to the MAGMA () linear algebra library, and can target NVIDIA's CUDA programming model or AMD's HIP programming model [45]. All data stored by this matrix implementation resides on the GPU at all times. The implementation currently supports a standard LAPACK column-major storage format as well as a low-storage format for block-diagonal matrices

$$\mathbf{A} = \begin{bmatrix} \mathbf{A_0} & 0 & \cdots & 0 \\ 0 & \mathbf{A_1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A_{n-1}} \end{bmatrix}.$$

This matrix implementation is best paired with the SUNLINEARSOLVER\_MAGMADENSE SUNLinearSolver.

The header file to include when using this module is sunmatrix\_magmadense.h. The installed library to link to is libsundials\_sunmatrixmagmadense.lib where .lib is typically .so for shared libraries and .a for static libraries.



The  ${\tt SUNMATRIX\_MAGMADENSE}$  module is experimental and subject to change.

# 8.8.1 SUNMATRIX\_MAGMADENSE functions

The SUNMATRIX\_MAGMADENSE module defines GPU-enabled implementations of all matrix operations listed in Section 8.1.1.

- 1. SUNMATGETID\_MagmaDense returns SUNMATRIX\_MAGMADENSE
- 2. SUNMatClone\_MagmaDense
- 3. SUNMatDestroy\_MagmaDense
- 4. SUNMatZero\_MagmaDense
- 5. SUNMatCopy\_MagmaDense
- 6. SUNMatScaleAdd\_MagmaDense
- 7. SUNMatScaleAddI\_MagmaDense
- $8. \ {\tt SUNMatMatvecSetup\_MagmaDense}$
- 9. SUNMatMatvec\_MagmaDense
- 10. SUNMatSpace\_MagmaDense

In addition, the SUNMATRIX\_MAGMADENSE module defines the following implementation specific functions:

# ${\tt SUNMatrix\_MagmaDense}$

Call A = SUNMatrix\_MagmaDense(M, N, memtype, memhelper, queue)

 $\ \, \text{Description} \quad \text{This constructor function creates and allocates memory for an } M \times N \, \text{\tt SUNMATRIX\_MAGMADENSE} \, \\$ 

SUNMatrix.

Arguments M (sunindextype) the number of matrix rows

N (sunindextype) the number of matrix columns

memtype (SUNMemoryType) the type of memory to use for the matrix data; can be

SUNMEMTYPE\_UVM or SUNMEMTYPE\_DEVICE.

memhelper (SUNMemoryHelper) the memory helper used for allocating data

queue a cudaStream\_t when using CUDA or a hipStream\_t when using HIP

Return value A SUNMatrix object if successful else NULL.

#### SUNMatrix\_MagmaDenseBlock

Call A = SUNMatrix\_MagmaDenseBlock(nblocks, M\_block, N\_block, memtype, memhelper,

queue)

Description This constructor function creates and allocates memory for a SUNMATRIX\_MAGMADENSE

SUNMatrix that is block diagonal with nblocks blocks of size  $M \times N$ .

Arguments nblocks (sunindextype) the number of matrix blocks

M\_block (sunindextype) the number of matrix rows in each block
N\_block (sunindextype) the number of matrix columns in each block

memtype (SUNMemoryType) the type of memory to use for the matrix data; can be

SUNMEMTYPE\_UVM or SUNMEMTYPE\_DEVICE.

memhelper (SUNMemoryHelper) the memory helper used for allocating data

queue a cudaStream\_t when using CUDA or a hipStream\_t when using HIP

Return value A SUNMatrix object if successful else NULL.

Notes The block diagonal format currently supports square matrices only.

#### SUNMatrix\_MagmaDense\_Rows

Call M = SUNMatrix\_MagmaDense\_Rows(A)

Description This function returns the rows dimension for the  $M \times N$  SUNMatrix. For block diagonal

matrices, this is computed as  $M_{\rm block} \times {\rm nblocks}$ .

Arguments A (SUNMatrix)

Return value The number of rows in the SUNMatrix.

#### SUNMatrix\_MagmaDense\_Columns

Call N = SUNMatrix\_MagmaDense\_Columns(A)

Description This function returns the columns dimension for the  $M \times N$  SUNMatrix. For block

diagonal matrices, this is computed as  $N_{\mathrm{block}} \times \mathrm{nblocks}$ .

Arguments A (SUNMatrix)

Return value The number of columns in the SUNMatrix.

#### SUNMatrix\_MagmaDense\_BlockRows

Call M = SUNMatrix\_MagmaDense\_BlockRows(A)

Description This function returns the number of rows in a block of the SUNMatrix.

Arguments A (SUNMatrix)

Return value The number of rows in a block of the SUNMatrix.

#### |SUNMatrix\_MagmaDense\_BlockColumns

Call N = SUNMatrix\_MagmaDense\_BlockColumns(A)

Description This function returns the number of columns in a block of the SUNMatrix.

Arguments A (SUNMatrix)

Return value The number of columns in a block of the SUNMatrix.

#### SUNMatrix\_MagmaDense\_LData

Call ldata = SUNMatrix\_MagmaDense\_LData(A)

Description This function returns the length of the data array for the SUNMatrix.

Arguments A (SUNMatrix)

Return value The length of the data array for the SUNMatrix.

#### SUNMatrix\_MagmaDense\_NumBlocks

Call nblocks = SUNMatrix\_MagmaDense\_NumBlocks(A)

Description This function returns the number of blocks in the SUNMatrix.

Arguments A (SUNMatrix)

Return value The number of matrix blocks.

#### SUNMatrix\_MagmaDense\_Data

Call data = SUNMatrix\_MagmaDense\_Data(A)

Description This function returns the SUNMatrix data array.

Arguments A (SUNMatrix)

Return value An array of pointers to the data arrays for each block in the SUNMatrix.

# SUNMatrix\_MagmaDense\_BlockData

Call data = SUNMatrix\_MagmaDense\_BlockData(A)

Description This function returns an array of pointers that point to the start of the data array for

each block.

Arguments A (SUNMatrix)

Return value An array of pointers to the data arrays for each block in the SUNMatrix.

#### SUNMatrix\_MagmaDense\_Block

Call data = SUNMatrix\_MagmaDense\_Block(A, k)

Description This function returns a pointer to the data for block k.

Arguments A (SUNMatrix)

Return value A pointer to the start of the data array for block k in the SUNMatrix.

Notes No bounds-checking is performed, k should be strictly less than nblocks.

## SUNMatrix\_MagmaDense\_Column

Call data = SUNMatrix\_MagmaDense\_Column(A, j)

Description This function returns a pointer to the data for column j of the matrix.

Arguments A (SUNMatrix)

Return value A pointer to the start of the data array for column j of the SUNMatrix.

Notes No bounds-checking is performed, j should be strictly less than nblocks  $*N_{block}$ .

#### SUNMatrix\_MagmaDense\_BlockColumn

Call data = SUNMatrix\_MagmaDense\_Column(A, k, j)

Description This function returns a pointer to the data for column j of block k.

Arguments A (SUNMatrix)

Return value A pointer to the start of the data array for column j of block k in the SUNMatrix.

Notes No bounds-checking is performed.

# |SUNMatrix\_MagmaDense\_CopyToDevice

Call retval = SUNMatrix\_MagmaDense\_CopyToDevice(A, h\_data)

Description This functions copies the matrix data to the GPU device from the provided host array.

Arguments A (SUNMatrix)

h\_data (realtype\*)

Return value SUNMAT\_SUCCESS if the copy operation was successful, or a nonzero error code otherwise

### |SUNMatrix\_MagmaDense\_CopyFromDevice

Call retval = SUNMatrix\_MagmaDense\_CopyFromDevice(A, h\_data)

Description This functions copies the matrix data from the GPU device to the provided host array.

Arguments A (SUNMatrix)

h\_data (realtype\*)

Return value SUNMAT\_SUCCESS if the copy operation was successful, or a nonzero error code otherwise

# 8.8.2 SUNMATRIX\_MAGMADENSE Usage Notes

When using the SUNMATRIX\_MAGMADENSE module with a SUNDIALS package (e.g. CVODE), the stream given to matrix should be the same stream used for the NVECTOR object that is provided to the package, and the NVECTOR object given to the SUNMatvec operation. If different streams are utilized, synchronization issues may occur.



# Chapter 9

# Description of the SUNLinearSolver module

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

The implementations provided with SUNDIALS work in coordination with the SUNDIALS generic NVECTOR and SUNMATRIX modules to provide a set of compatible data structures and solvers for the solution of linear systems using direct or iterative (matrix-based or matrix-free) methods. Moreover, advanced users can provide a customized SUNLinearSolver implementation to any SUNDIALS package, particularly in cases where they provide their own NVECTOR and/or SUNMATRIX modules.

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

$$\tilde{A}\tilde{x} = \tilde{b} \tag{9.1}$$

where

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

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

$$\tilde{x} = S_2 P_2 x,$$
(9.2)

and where

- $P_1$  is the left preconditioner,
- $P_2$  is the right preconditioner,
- $S_1$  is a diagonal matrix of scale factors for  $P_1^{-1}b$ ,
- $S_2$  is a diagonal matrix of scale factors for  $P_2x$ .

The scaling matrices are chosen so that  $S_1P_1^{-1}b$  and  $S_2P_2x$  have dimensionless components. If preconditioning is done on the left only  $(P_2 = I)$ , by a matrix P, then  $S_2$  must be a scaling for x, while  $S_1$  is a scaling for  $P^{-1}b$ , and so may also be taken as a scaling for x. Similarly, if preconditioning is done on the right only  $(P_1 = I \text{ and } P_2 = P)$ , then  $S_1$  must be a scaling for b, while  $S_2$  is a scaling for Px, and may also be taken as a scaling for b.

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

$$\left\|\tilde{b} - \tilde{A}\tilde{x}\right\|_{2} < \text{tol.}$$

When provided an iterative SUNLINSOL implementation that does not support the scaling matrices  $S_1$  and  $S_2$ , SUNDIALS' packages will adjust the value of tol accordingly (see §9.4.2 for more details). In this case, they instead request that iterative linear solvers stop based on the criteria

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

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

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

For users interested in providing their own SUNLINSOL module, the following section presents the SUNLINSOL API and its implementation beginning with the definition of SUNLINSOL functions in sections 9.1.1 – 9.1.3. This is followed by the definition of functions supplied to a linear solver implementation in section 9.1.4. A table of linear solver return codes is given in section 9.1.5. The SUNLinearSolver type and the generic SUNLINSOL module are defined in section 9.1.6. The section 9.2 discusses compatibility between the SUNDIALS-provided SUNLINSOL modules and SUNMATRIX modules. Section 9.3 lists the requirements for supplying a custom SUNLINSOL module and discusses some intended use cases. Users wishing to supply their own SUNLINSOL module are encouraged to use the SUNLINSOL implementations provided with SUNDIALS as a template for supplying custom linear solver modules. The SUNLINSOL functions required by this SUNDIALS package as well as other package specific details are given in section 9.4. The remaining sections of this chapter present the SUNLINSOL modules provided with SUNDIALS.

# 9.1 The SUNLinear Solver API

The SUNLINSOL API defines several linear solver operations that enable SUNDIALS packages to utilize any SUNLINSOL implementation that provides the required functions. These functions can be divided into three categories. The first are the core linear solver functions. The second group of functions consists of set routines to supply the linear solver with functions provided by the SUNDIALS time integrators and to modify solver parameters. The final group consists of get routines for retrieving linear solver statistics. All of these functions are defined in the header file sundials/sundials\_linearsolver.h.

#### 9.1.1 SUNLinearSolver core functions

The core linear solver functions consist of two required functions to get the linear solver type (SUNLinSolGetType) and solve the linear system Ax = b (SUNLinSolSolve). The remaining functions are for getting the solver ID (SUNLinSolGetID), initializing the linear solver object once all solver-specific options have been set (SUNLinSolInitialize), setting up the linear solver object to utilize an updated matrix A (SUNLinSolSetup), and for destroying the linear solver object (SUNLinSolFree) are optional.

## SUNLinSolGetType

Call type = SUNLinSolGetType(LS);

Description The required function SUNLinSolGetType returns the type identifier for the linear solver

LS. It is used to determine the solver type (direct, iterative, or matrix-iterative) from

the abstract  ${\tt SUNLinearSolver}$  interface.

Arguments LS (SUNLinearSolver) a SUNLINSOL object.

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

- SUNLINEARSOLVER\_DIRECT 0, the SUNLINSOL module requires a matrix, and computes an 'exact' solution to the linear system defined by that matrix.
- SUNLINEARSOLVER\_ITERATIVE 1, the SUNLINSOL module does not require a matrix (though one may be provided), and computes an inexact solution to the linear system using a matrix-free iterative algorithm. That is it solves the linear system defined by the package-supplied ATimes routine (see SUNLinSolSetATimes below), even if that linear system differs from the one encoded in the matrix object (if one is provided). As the solver computes the solution only inexactly (or may diverge), the linear solver should check for solution convergence/accuracy as appropriate.
- SUNLINEARSOLVER\_MATRIX\_ITERATIVE 2, the SUNLINSOL module requires a matrix, and computes an inexact solution to the linear system defined by that matrix using an iterative algorithm. That is it solves the linear system defined by the matrix object even if that linear system differs from that encoded by the package-supplied ATimes routine. As the solver computes the solution only inexactly (or may diverge), the linear solver should check for solution convergence/accuracy as appropriate.

Notes See section 9.3.1 for more information on intended use cases corresponding to the linear solver type.

F2003 Name FSUNLinSolGetType

#### SUNLinSolGetID

Call id = SUNLinSolGetID(LS);

Description The optional function SUNLinSolGetID returns the identifier for the linear solver LS.

Arguments LS (SUNLinearSolver) a SUNLINSOL object.

Return value The return value id (of type int) will be a non-negative value defined by the enumer-

ation SUNLinearSolver ID.

Notes It is recommended that a user-supplied SUNLinearSolver return the

SUNLINEARSOLVER\_CUSTOM identifier.

F2003 Name FSUNLinSolGetID

#### SUNLinSolInitialize

Call retval = SUNLinSolInitialize(LS);

Description The optional function SUNLinSolInitialize performs linear solver initialization (as-

suming that all solver-specific options have been set).

Arguments LS (SUNLinearSolver) a SUNLINSOL object.

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

returning one of the generic error codes listed in Table 9.1.

 $F2003\ \mathrm{Name}\ FSUNLinSolInitialize$ 

#### SUNLinSolSetup

Call retval = SUNLinSolSetup(LS, A);

Description

The *optional* function SUNLinSolSetup performs any linear solver setup needed, based on an updated system SUNMATRIX A. This may be called frequently (e.g., with a full Newton method) or infrequently (for a modified Newton method), based on the type of integrator and/or nonlinear solver requesting the solves.

Arguments LS (SUNLinearSolver) a SUNLINSOL object.

A (SUNMatrix) a SUNMATRIX object.

Return value This should return zero for a successful call, a positive value for a recoverable failure and a negative value for an unrecoverable failure, ideally returning one of the generic error codes listed in Table 9.1.

F2003 Name FSUNLinSolSetup

# SUNLinSolSolve

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

Description The required function SUNLinSolSolve solves a linear system Ax = b.

Arguments LS (SUNLinearSolver) a SUNLINSOL object.

A (SUNMatrix) a SUNMATRIX object.

x (N\_Vector) a NVECTOR object containing the initial guess for the solution of the linear system, and the solution to the linear system upon return.

b (N\_Vector) a NVECTOR object containing the linear system right-hand side.

tol (realtype) the desired linear solver tolerance.

Return value This should return zero for a successful call, a positive value for a recoverable failure and a negative value for an unrecoverable failure, ideally returning one of the generic error codes listed in Table 9.1.

Notes **Direct solvers:** can ignore the tol argument.

Matrix-free solvers: (those that identify as SUNLINEARSOLVER\_ITERATIVE) can ignore the SUNMATRIX input A, and should instead rely on the matrix-vector product function supplied through the routine SUNLinSolSetATimes.

Iterative solvers: (those that identify as SUNLINEARSOLVER\_ITERATIVE or SUNLINEARSOLVER\_MATRIX\_ITERATIVE) should attempt to solve to the specified tolerance tol in a weighted 2-norm. If the solver does not support scaling then it should just use a 2-norm.

F2003 Name FSUNLinSolSolve

# SUNLinSolFree

Call retval = SUNLinSolFree(LS);

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

Arguments LS (SUNLinearSolver) a SUNLINSOL object.

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

F2003 Name FSUNLinSolFree

#### 9.1.2 SUNLinearSolver set functions

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

#### SUNLinSolSetATimes

Call retval = SUNLinSolSetATimes(LS, A\_data, ATimes);

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

This routine provides an ATimesFn function pointer, as well as a void\* pointer to a data structure used by this routine, to a linear solver object. SUNDIALS packages will call this function to set the matrix-vector product function to either a solver-provided difference-quotient via vector operations or a user-supplied solver-specific routine.

Arguments LS (SUNLinearSolver) a SUNLINSOL object.

A\_data (void\*) data structure passed to ATimes.

ATimes (ATimesFn) function pointer implementing the matrix-vector product routine.

Return value This routine should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 9.1.

 $F2003 \ \mathrm{Name}$  FSUNLinSolSetATimes

#### SUNLinSolSetPreconditioner

Call retval = SUNLinSolSetPreconditioner(LS, Pdata, Pset, Psol);

Description The optional function SUNLinSolSetPreconditioner provides PSetupFn and PSolveFn function pointers that implement the preconditioner solves  $P_1^{-1}$  and  $P_2^{-1}$  from equations (9.1)-(9.2). This routine will be called by a SUNDIALS package, which will provide translation between the generic Pset and Psol calls and the package- or user-supplied routines.

Arguments LS (SUNLinearSolver) a SUNLINSOL object.

Pdata (void\*) data structure passed to both Pset and Psol.

Pset (PSetupFn) function pointer implementing the preconditioner setup.

Psol (PSolveFn) function pointer implementing the preconditioner solve.

Return value This routine should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 9.1.

F2003 Name FSUNLinSolSetPreconditioner

# SUNLinSolSetScalingVectors

Call retval = SUNLinSolSetScalingVectors(LS, s1, s2);

Description The optional function SUNLinSolSetScalingVectors provides left/right scaling vectors for the linear system solve. Here, s1 and s2 are NVECTOR of positive scale factors containing the diagonal of the matrices  $S_1$  and  $S_2$  from equations (9.1)-(9.2), respectively. Neither of these vectors need to be tested for positivity, and a NULL argument for either

indicates that the corresponding scaling matrix is the identity.

Arguments LS (SUNLinearSolver) a SUNLINSOL object.

s1 (N\_Vector) diagonal of the matrix  $S_1$ 

s2 (N\_Vector) diagonal of the matrix  $S_2$ 

Return value This routine should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 9.1.

F2003 Name FSUNLinSolSetScalingVectors

# 9.1.3 SUNLinearSolver get functions

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

#### SUNLinSolNumIters

Call its = SUNLinSolNumIters(LS);

Description The optional function SUNLinSolNumIters should return the number of linear iterations

performed in the last 'solve' call.

Arguments LS (SUNLinearSolver) a SUNLINSOL object.

Return value int containing the number of iterations

F2003 Name FSUNLinSolNumIters

#### SUNLinSolResNorm

Description The optional function SUNLinSolResNorm should return the final residual norm from

the last 'solve' call.

Arguments LS (SUNLinearSolver) a SUNLINSOL object.

Return value realtype containing the final residual norm

F2003 Name FSUNLinSolResNorm

#### SUNLinSolResid

Call rvec = SUNLinSolResid(LS);

Description If an iterative method computes the preconditioned initial residual and returns with

a successful solve without performing any iterations (i.e., either the initial guess or the preconditioner is sufficiently accurate), then this *optional* routine may be called by the SUNDIALS package. This routine should return the NVECTOR containing the

preconditioned initial residual vector.

Arguments LS (SUNLinearSolver) a SUNLINSOL object.

Return value N\_Vector containing the final residual vector

Notes Since N\_Vector is actually a pointer, and the results are not modified, this routine

should not require additional memory allocation. If the SUNLINSOL object does not retain a vector for this purpose, then this function pointer should be set to NULL in the

implementation.

F2003 Name FSUNLinSolResid

#### SUNLinSolLastFlag

Call lflag = SUNLinSolLastFlag(LS);

Description The optional function SUNLinSolLastFlag should return the last error flag encountered

within the linear solver. This is not called by the SUNDIALS packages directly; it allows

the user to investigate linear solver issues after a failed solve.

Arguments LS (SUNLinearSolver) a SUNLINSOL object.

Return value sunindextype containing the most recent error flag

F2003 Name FSUNLinSollastFlag

#### SUNLinSolSpace

Call retval = SUNLinSolSpace(LS, &lrw, &liw);

Description The optional function SUNLinSolSpace should return the storage requirements for the

linear solver LS.

Arguments LS (SUNLinearSolver) a SUNLINSOL object.

lrw (long int\*) the number of realtype words stored by the linear solver.

liw (long int\*) the number of integer words stored by the linear solver.

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

returning one of the generic error codes listed in Table 9.1.

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

 $F2003 \ \mathrm{Name} \ FSUNLinSolSpace$ 

# 9.1.4 Functions provided by SUNDIALS packages

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

#### ATimesFn

Definition typedef int (\*ATimesFn)(void \*A\_data, N\_Vector v, N\_Vector z);

Purpose These functions compute the action of a matrix on a vector, performing the operation

z=Av. Memory for **z** should already be allocted prior to calling this function. The

vector v should be left unchanged.

Arguments A\_data is a pointer to client data, the same as that supplied to SUNLinSolSetATimes.

v is the input vector to multiply.

z is the output vector computed.

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

#### PSetupFn

Definition typedef int (\*PSetupFn)(void \*P\_data)

Purpose These functions set up any requisite problem data in preparation for calls to the corre-

sponding PSolveFn.

Arguments P\_data is a pointer to client data, the same pointer as that supplied to the routine

 ${\tt SUNLinSolSetPreconditioner}.$ 

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

PSolveFn

Definition typedef int (\*PSolveFn)(void \*P\_data, N\_Vector r, N\_Vector z, realtype tol, int lr)

Purpose

These functions solve the preconditioner equation Pz = r for the vector z. Memory for z should already be allocted prior to calling this function. The parameter P\_data is a pointer to any information about P which the function needs in order to do its job (set up by the corresponding PSetupFn). The parameter 1r is input, and indicates whether P is to be taken as the left preconditioner or the right preconditioner: lr = 1 for left and 1r = 2 for right. If preconditioning is on one side only, 1r can be ignored. If the preconditioner is iterative, then it should strive to solve the preconditioner equation so that

$$||Pz - r||_{\text{wrms}} < tol$$

where the weight vector for the WRMS norm may be accessed from the main package memory structure. The vector r should not be modified by the PSolveFn.

Arguments

P\_data is a pointer to client data, the same pointer as that supplied to the routine SUNLinSolSetPreconditioner.

is the right-hand side vector for the preconditioner system. r

is the solution vector for the preconditioner system. z

is the desired tolerance for an iterative preconditioner. tol

is flag indicating whether the routine should perform left (1) or right (2) prelr conditioning.

Return value This routine should return 0 if successful and a non-zero value if unsuccessful. On a failure, a negative return value indicates an unrecoverable condition, while a positive value indicates a recoverable one, in which the calling routine may reattempt the solution after updating preconditioner data.

#### 9.1.5SUNLinearSolver return codes

The functions provided to SUNLINSOL modules by each SUNDIALS package, and functions within the SUNDIALS-provided SUNLINSOL implementations utilize a common set of return codes, shown in Table 9.1. These adhere to a common pattern: 0 indicates success, a postitive value corresponds to a recoverable failure, and a negative value indicates a non-recoverable failure. Aside from this pattern, the actual values of each error code are primarily to provide additional information to the user in case of a linear solver failure.

Table 9.1: Description of the SUNLinearSolver error codes

Name	Value	Description	
SUNLS_SUCCESS	0	successful call or converged solve	
SUNLS_MEM_NULL	-801	the memory argument to the function is NULL	
SUNLS_ILL_INPUT	-802	an illegal input has been provided to the function	
SUNLS_MEM_FAIL	-803	failed memory access or allocation	
SUNLS_ATIMES_NULL	-804	the Atimes function is NULL	
		continued on next page	

Name	Value	Description	
SUNLS_ATIMES_FAIL_UNREC	-805	an unrecoverable failure occurred in the ATimes routine	
SUNLS_PSET_FAIL_UNREC	-806	an unrecoverable failure occurred in the Pset routine	
SUNLS_PSOLVE_NULL	-807	the preconditioner solve function is NULL	
SUNLS_PSOLVE_FAIL_UNREC	-808	an unrecoverable failure occurred in the Psolve routine	
SUNLS_PACKAGE_FAIL_UNREC	-809	an unrecoverable failure occurred in an external linear solver package	
SUNLS_GS_FAIL	-810	a failure occurred during Gram-Schmidt orthogonalization (SUNLINSOL_SPGMR/SUNLINSOL_SPFGMR)	
SUNLS_QRSOL_FAIL	-811	a singular $R$ matrix was encountered in a QR factorization (SUNLINSOL_SPGMR/SUNLINSOL_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 (SUNLINSOL_SPGMR/SUNLINSOL_SPFGMR)	
SUNLS_LUFACT_FAIL	808	a singular matrix was encountered during a LU factorization (SUNLINSOL_DENSE/SUNLINSOL_BAND)	

# 9.1.6 The generic SUNLinearSolver module

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

```
typedef struct _generic_SUNLinearSolver *SUNLinearSolver;
struct _generic_SUNLinearSolver {
  void *content;
  struct _generic_SUNLinearSolver_Ops *ops;
};
```

where the \_generic\_SUNLinearSolver\_Ops structure is a list of pointers to the various actual linear solver operations provided by a specific implementation. The \_generic\_SUNLinearSolver\_Ops structure is defined as

```
(*setscalingvectors)(SUNLinearSolver,
  int
                                              N_Vector, N_Vector);
  int
                        (*initialize)(SUNLinearSolver);
  int
                        (*setup)(SUNLinearSolver, SUNMatrix);
  int
                        (*solve)(SUNLinearSolver, SUNMatrix, N_Vector,
                                 N_Vector, realtype);
  int
                        (*numiters)(SUNLinearSolver);
  realtype
                        (*resnorm)(SUNLinearSolver);
                        (*lastflag)(SUNLinearSolver);
  sunindxetype
                        (*space)(SUNLinearSolver, long int*, long int*);
                        (*resid)(SUNLinearSolver);
  N Vector
  int
                        (*free)(SUNLinearSolver);
};
```

The generic SUNLINSOL module defines and implements the linear solver operations defined in Sections 9.1.1-9.1.3. These routines are in fact only wrappers to the linear solver operations defined by a particular SUNLINSOL implementation, which are accessed through the *ops* field of the SUNLinearSolver structure. To illustrate this point we show below the implementation of a typical linear solver operation from the generic SUNLINSOL module, namely SUNLinSolInitialize, which initializes a SUNLINSOL object for use after it has been created and configured, and returns a flag denoting a successful/failed operation:

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

The Fortran 2003 interface provides a bind(C) derived-type for the \_generic\_SUNLinearSolver and the \_generic\_SUNLinearSolver\_Ops structures. Their definition is given below.

```
type, bind(C), public :: SUNLinearSolver
type(C_PTR), public :: content
type(C_PTR), public :: ops
end type SUNLinearSolver
type, bind(C), public :: SUNLinearSolver_Ops
type(C_FUNPTR), public :: gettype
type(C_FUNPTR), public :: setatimes
type(C_FUNPTR), public :: setpreconditioner
type(C_FUNPTR), public :: setscalingvectors
type(C_FUNPTR), public :: initialize
type(C_FUNPTR), public :: setup
type(C_FUNPTR), public :: solve
type(C_FUNPTR), public :: numiters
type(C_FUNPTR), public :: resnorm
type(C_FUNPTR), public :: lastflag
type(C_FUNPTR), public :: space
type(C_FUNPTR), public :: resid
type(C_FUNPTR), public :: free
end type SUNLinearSolver_Ops
```

# 9.2 Compatibility of SUNLinearSolver modules

We note that not all SUNLINSOL types are compatible with all SUNMATRIX and NVECTOR types provided with SUNDIALS. In Table 9.2 we show the matrix-based linear solvers available as SUNLINSOL modules,

and the compatible matrix implementations. Recall that Table 4.1 shows the compatibility between all SUNLINSOL modules and vector implementations.

Table 9.2:	SUNDIALS	matrix-based	linear	solvers	and	matrix	implementations	that	$\operatorname{can}$	be	used	for
each.												

Linear Solver	Dense	Banded	Sparse	SLUNRloc	User
Interface	Matrix	Matrix	Matrix	Matrix	Supplied
Dense	✓				✓
Band		✓			✓
LapackDense	✓				✓
LapackBand		✓			✓
KLU			<b>✓</b>		✓
SuperLU_DIST				<b>✓</b>	✓
SUPERLUMT			<b>√</b>		<b>√</b>
User supplied	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>	<b>√</b>

# 9.3 Implementing a custom SUNLinearSolver module

A particular implementation of the Sunlinsol module must:

- Specify the *content* field of the SUNLinearSolver object.
- Define and implement a minimal subset of the linear solver operations. See the section 9.4 to determine which SUNLINSOL operations are required for this SUNDIALS package.

Note that the names of these routines should be unique to that implementation in order to permit using more than one SUNLINSOL module (each with different SUNLinearSolver internal data representations) in the same code.

• Define and implement user-callable constructor and destructor routines to create and free a SUNLinearSolver with the new *content* field and with *ops* pointing to the new linear solver operations.

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

To aid in the creation of custom SUNLINSOL modules the generic SUNLINSOL module provides the utility functions SUNLinSolNewEmpty and SUNLinSolFreeEmpty. When used in custom SUNLINSOL constructors the function SUNLinSolNewEmpty will ease the introduction of any new optional linear solver operations to the SUNLINSOL API by ensuring only required operations need to be set.

# SUNLinSolNewEmpty

Call LS = SUNLinSolNewEmpty();

Description The function SUNLinSolNewEmpty allocates a new generic SUNLINSOL object and initializes its content pointer and the function pointers in the operations structure to NULL.

Arguments None

Return value This function returns a SUNLinearSolver object. If an error occurs when allocating the object, then this routine will return NULL.

F2003 Name FSUNLinSolNewEmpty

# SUNLinSolFreeEmpty

Call SUNLinSolFreeEmpty(LS);

Description This routine frees the generic SUNLinSolFreeEmpty object, under the assumption that

any implementation-specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the ops pointer is NULL,

and, if it is not, it will free it as well.

Arguments LS (SUNLinearSolver)

Return value None

F2003 Name FSUNLinSolFreeEmpty

Additionally, a SUNLINSOL implementation may do the following:

- Define and implement additional user-callable "set" routines acting on the SUNLinearSolver, e.g., for setting various configuration options to tune the linear solver to a particular problem.
- Provide additional user-callable "get" routines acting on the SUNLinearSolver object, e.g., for returning various solve statistics.

#### 9.3.1 Intended use cases

The SUNLINSOL (and SUNMATRIX) APIs are designed to require a minimal set of routines to ease interfacing with custom or third-party linear solver libraries. External solvers provide similar routines with the necessary functionality and thus will require minimal effort to wrap within custom SUNMATRIX and SUNLINSOL implementations. Sections 8.2 and 9.4 include a list of the required set of routines that compatible SUNMATRIX and SUNLINSOL implementations must provide. As SUNDIALS packages utilize generic SUNLINSOL modules allowing for user-supplied SUNLinearSolver implementations, there exists a wide range of possible linear solver combinations. Some intended use cases for both the SUNDIALS-provided and user-supplied SUNLINSOL modules are discussed in the following sections.

#### Direct linear solvers

Direct linear solver modules require a matrix and compute an 'exact' solution to the linear system defined by the matrix. Multiple matrix formats and associated direct linear solvers are supplied with SUNDIALS through different SUNMATRIX and SUNLINSOL implementations. SUNDIALS packages strive to amortize the high cost of matrix construction by reusing matrix information for multiple nonlinear iterations. As a result, each package's linear solver interface recomputes Jacobian information as infrequently as possible.

Alternative matrix storage formats and compatible linear solvers that are not currently provided by, or interfaced with, SUNDIALS can leverage this infrastructure with minimal effort. To do so, a user must implement custom SUNMATRIX and SUNLINSOL wrappers for the desired matrix format and/or linear solver following the APIs described in Chapters 8 and 9. This user-supplied SUNLINSOL module must then self-identify as having SUNLINEARSOLVER\_DIRECT type.

#### Matrix-free iterative linear solvers

Matrix-free iterative linear solver modules do not require a matrix and compute an inexact solution to the linear system defined by the package-supplied ATimes routine. SUNDIALS supplies multiple scaled, preconditioned iterative linear solver (spils) SUNLINSOL modules that support scaling to allow users to handle non-dimensionalization (as best as possible) within each SUNDIALS package and retain variables and define equations as desired in their applications. For linear solvers that do not support left/right scaling, the tolerance supplied to the linear solver is adjusted to compensate (see section 9.4.2 for more details); however, this use case may be non-optimal and cannot handle situations where the magnitudes of different solution components or equations vary dramatically within a single problem.

To utilize alternative linear solvers that are not currently provided by, or interfaced with, SUNDI-ALS a user must implement a custom SUNLINSOL wrapper for the linear solver following the API described in Chapter 9. This user-supplied sunlinesol module must then self-identify as having sunlinear solver\_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 cost of matrix construction. As in the direct linear solver case, only wrappers for the matrix and linear solver in Sunmatrix and Sunlinsol implementations need to be created to utilize a new linear solver. This user-supplied Sunlinsol module must then self-identify as having Sunlinear Solver 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 that 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 module where the matrix is *purely* for preconditioning and the linear system is defined by the package-supplied ATimes routine, we envision two current possibilities.

The preferred approach is for users to employ one of the SUNDIALS spils SUNLINSOL implementations (SUNLINSOL\_SPGMR, SUNLINSOL\_SPFGMR, SUNLINSOL\_SPECGS, SUNLINSOL\_SPTFQMR, or SUNLINSOL\_PCG) as the outer solver. The creation and storage of the preconditioner matrix, and interfacing with the corresponding linear solver, can be handled through a package's preconditioner 'setup' and 'solve' functionality (see §4.5.8.2) without creating SUNMATRIX and SUNLINSOL implementations. This usage mode is recommended primarily because the SUNDIALS-provided spils modules support the scaling as described above.

A second approach supported by the linear solver APIs is as follows. If the SUNLINSOL implementation is matrix-based, self-identifies as having SUNLINEARSOLVER\_ITERATIVE type, and also provides a non-NULL SUNLinSolSetATimes routine, then each SUNDIALS package will call that routine to attach its package-specific matrix-vector product routine to the SUNLINSOL object. The SUNDIALS package will then call the SUNLINSOL-provided SUNLinSolSetup routine (infrequently) to update matrix information, but will provide current matrix-vector products to the SUNLINSOL implementation through the package-supplied ATimesFn routine.

# 9.4 IDA SUNLinearSolver interface

Table 9.3 below lists the SUNLINSOL 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  $\checkmark$  to indicate that they are required, or with  $\dagger$  to indicate that they are only called if they are non-NULL in the SUNLINSOL 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.

Since there are a wide range of potential SUNLINSOL use cases, the following subsections describe some details of the IDALS interface, in the case that interested users wish to develop custom SUNLINSOL modules.

	DIRECT	ITERATIVE	MATRIX_ITERATIVE
SUNLinSolGetType	<b>√</b>	<b>√</b>	<b>√</b>
SUNLinSolSetATimes	†	<b>√</b>	†
SUNLinSolSetPreconditioner	†	†	†
SUNLinSolSetScalingVectors	†	†	†
SUNLinSolInitialize	<b>√</b>	<b>√</b>	<b>√</b>
SUNLinSolSetup	<b>√</b>	<b>√</b>	<b>√</b>
SUNLinSolSolve	<b>√</b>	<b>√</b>	<b>√</b>
SUNLinSolNumIters		<b>√</b>	<b>√</b>
SUNLinSolResid		<b>√</b>	<b>√</b>
$^1$ SUNLinSolLastFlag			
$^2$ SUNLinSolFree			
SUNLinSolSpace	†	†	†

Table 9.3: List of linear solver function usage in the IDALS interface

# 9.4.1 Lagged matrix information

If the SUNLINSOL object self-identifies as having type SUNLINEARSOLVER\_DIRECT or SUNLINEARSOLVER\_MATRIX\_ITERATIVE, then the SUNLINSOL object solves a linear system defined by a SUNMATRIX object. IDALS will update the matrix information infrequently according to the strategies outlined in §2.1. To this end, we differentiate between the desired linear system Mx=b and the actual linear system  $\bar{M}\bar{x}=b$ . Since IDALS updates the SUNMATRIX object infrequently, it is likely that  $\alpha \neq \bar{\alpha}$ , and in turn  $M \neq \bar{M}$ . Therefore, after calling the SUNLINSOL-provided SUNLinSolSolve routine, we test whether  $\alpha/\bar{\alpha} \neq 1$ , and if this is the case we scale the solution  $\bar{x}$  to correct the linear system solution x via

$$x = \frac{2}{1 + \alpha/\bar{\alpha}}\bar{x}.\tag{9.3}$$

The motivation for this selection of the scaling factor  $c = 2/(1 + \alpha/\bar{\alpha})$  is discussed in detail in [11, 31]. In short, if we consider a stationary iteration for the linear system as consisting of a solve with  $\bar{M}$  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  $E = I - c\bar{M}^{-1}M$ , 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.

#### 9.4.2 Iterative linear solver tolerance

If the SUNLINSOL 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

$$W_i \approx W_{mean}$$
, for  $i = 0, \dots, n-1$ .

2. The SUNLINSOL object uses a standard 2-norm to measure convergence.

Since IDA uses identical left and right scaling matrices,  $S_1 = S_2 = S = \text{diag}(W)$ , then the linear solver convergence requirement is converted as follows (using the notation from equations (9.1)-(9.2)):

$$\begin{split} \left\| \tilde{b} - \tilde{A} \tilde{x} \right\|_2 < \text{tol} \\ \Leftrightarrow \quad \left\| S P_1^{-1} b - S P_1^{-1} A x \right\|_2 < \text{tol} \\ \Leftrightarrow \quad \sum_{i=0}^{n-1} \left[ W_i \left( P_1^{-1} (b - A x) \right)_i \right]^2 < \text{tol}^2 \\ \Leftrightarrow \quad W_{mean}^2 \sum_{i=0}^{n-1} \left[ \left( P_1^{-1} (b - A x) \right)_i \right]^2 < \text{tol}^2 \\ \Leftrightarrow \quad \sum_{i=0}^{n-1} \left[ \left( P_1^{-1} (b - A x) \right)_i \right]^2 < \left( \frac{\text{tol}}{W_{mean}} \right)^2 \\ \Leftrightarrow \quad \left\| P_1^{-1} (b - A x) \right\|_2 < \frac{\text{tol}}{W_{mean}} \end{split}$$

Therefore the tolerance scaling factor

$$W_{mean} = ||W||_2/\sqrt{n}$$

is computed and the scaled tolerance  $delta = tol/W_{mean}$  is supplied to the SUNLINSOL object.

# 9.5 The SUNLinearSolver\_Dense implementation

This section describes the SUNLINSOL implementation for solving dense linear systems. The SUNLINSOL\_DENSE module is designed to be used with the corresponding SUNMATRIX\_DENSE matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR\_SERIAL, NVECTOR\_OPENMP, or NVECTOR\_PTHREADS).

To access the SUNLINSOL\_DENSE module, include the header file sunlinsol/sunlinsol\_dense.h. We note that the SUNLINSOL\_DENSE module is accessible from SUNDIALS packages without separately linking to the libsundials\_sunlinsoldense module library.

# 9.5.1 SUNLinearSolver\_Dense description

This solver is constructed to perform the following operations:

- The "setup" call performs a LU factorization with partial (row) pivoting  $(\mathcal{O}(N^3) \cos t)$ , 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  $(\mathcal{O}(N^2) \text{ cost})$ .

# 9.5.2 SUNLinearSolver\_Dense functions

The SUNLINSOL\_DENSE module provides the following user-callable constructor for creating a SUNLinearSolver object.

#### SUNLinSol\_Dense

Call LS = SUNLinSol\_Dense(y, A);

Description The function SUNLinSol\_Dense creates and allocates memory for a dense

SUNLinearSolver object.

Arguments y (N\_Vector) a template for cloning vectors needed within the solver

A (SUNMatrix) a  ${\tt SUNMATRIX\_DENSE}$  matrix template for cloning matrices needed

within the solver

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

routine will return NULL.

Notes This routine will perform consistency checks to ensure that it is called with con-

sistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX\_DENSE matrix type and the NVECTOR\_SERIAL, NVECTOR\_OPENMP, and NVECTOR\_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this

compatibility check.

Deprecated Name For backward compatibility, the wrapper function SUNDenseLinearSolver with

idential input and output arguments is also provided.

F2003 Name FSUNLinSol\_Dense

The SUNLINSOL\_DENSE module defines implementations of all "direct" linear solver operations listed in Sections 9.1.1 - 9.1.3:

• SUNLinSolGetType\_Dense

- SUNLinSolInitialize\_Dense this does nothing, since all consistency checks are performed at solver creation.
- SUNLinSolSetup\_Dense this performs the LU factorization.
- SUNLinSolSolve\_Dense this uses the LU factors and pivots array to perform the solve.
- SUNLinSolLastFlag\_Dense
- SUNLinSolSpace\_Dense this only returns information for the storage within the solver object, i.e. storage for N, last\_flag, and pivots.
- SUNLinSolFree\_Dense

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an 'F' to the function name.

# 9.5.3 SUNLinearSolver\_Dense Fortran interfaces

The SUNLINSOL\_DENSE module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

#### FORTRAN 2003 interface module

The fsunlinsol\_dense\_mod FORTRAN module defines interfaces to all SUNLINSOL\_DENSE C functions using the intrinsic iso\_c\_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function SUNLinSol\_Dense is interfaced as FSUNLinSol\_Dense.

The Fortran 2003 sunlinsol\_dense\_mod, and linking to the library libsundials\_fsunlinsoldense\_mod. lib in addition to the C library. For details on where the library and module file fsunlinsol\_dense\_mod.mod are installed see Appendix A. We note that the module is accessible from the Fortran 2003 sundials integrators without separately linking to the libsundials\_fsunlinsoldense\_mod library.

#### FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL\_DENSE module also includes a Fortran-callable function for creating a SUNLinearSolver object.

#### FSUNDENSELINSOLINIT

Call FSUNDENSELINSOLINIT(code, ier)

Description The function FSUNDENSELINSOLINIT can be called for Fortran programs to create a

dense SUNLinearSolver object.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for

KINSOL, and 4 for ARKODE).

Return value ier is a return completion flag equal to 0 for a success return and -1 otherwise. See

printed message for details in case of failure.

Notes This routine must be called *after* both the NVECTOR and SUNMATRIX objects have been

initialized.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL\_DENSE module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

#### FSUNMASSDENSELINSOLINIT

Call FSUNMASSDENSELINSOLINIT(ier)

Description The function FSUNMASSDENSELINSOLINIT can be called for Fortran programs to create

a dense SUNLinearSolver object for mass matrix linear systems.

Arguments None

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes This routine must be called after both the NVECTOR and SUNMATRIX mass-matrix

objects have been initialized.

#### 9.5.4 SUNLinearSolver Dense content

The  $SUNLINSOL\_DENSE$  module defines the content field of a SUNLinearSolver as the following structure:

```
struct _SUNLinearSolverContent_Dense {
   sunindextype N;
   sunindextype *pivots;
   sunindextype last_flag;
};
```

These entries of the *content* field contain the following information:

N - size of the linear system,

pivots - index array for partial pivoting in LU factorization,

last\_flag - last error return flag from internal function evaluations.

# 9.6 The SUNLinearSolver\_Band implementation

This section describes the SUNLINSOL implementation for solving banded linear systems. The SUNLINSOL\_BAND module is designed to be used with the corresponding SUNMATRIX\_BAND matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR\_SERIAL, NVECTOR\_OPENMP, or NVECTOR\_PTHREADS).

To access the SUNLINSOL\_BAND module, include the header file sunlinsol/sunlinsol\_band.h. We note that the SUNLINSOL\_BAND module is accessible from SUNDIALS packages without separately linking to the libsundials\_sunlinsolband module library.

# 9.6.1 SUNLinearSolver\_Band description

This solver is constructed to perform the following operations:

- The "setup" call performs a LU factorization with partial (row) pivoting, PA = LU, where P is a permutation matrix, L is a lower triangular matrix with 1's on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX\_BAND object A, with pivoting information encoding P stored in the pivots array.
- The "solve" call performs pivoting and forward and backward substitution using the stored pivots array and the LU factors held in the SUNMATRIX\_BAND object.
- A must be allocated to accommodate the increase in upper bandwidth that occurs during factorization. More precisely, if A is a band matrix with upper bandwidth mu and lower bandwidth 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.



#### 9.6.2 SUNLinearSolver Band functions

The SUNLINSOL\_BAND module provides the following user-callable constructor for creating a SUNLinearSolver object.

CITITIT	inSol	D 1
	וחאחו	Rand

Call LS = SUNLinSol\_Band(y, A);

Description The function SUNLinSol\_Band creates and allocates memory for a band

SUNLinearSolver object.

Arguments y (N\_Vector) a template for cloning vectors needed within the solver

A (SUNMatrix) a SUNMATRIX\_BAND matrix template for cloning matrices needed

within the solver

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

routine will return NULL.

Notes This routine will perform consistency checks to ensure that it is called with con-

sistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX\_BAND matrix type and the NVECTOR\_SERIAL, NVECTOR\_OPENMP, and NVECTOR\_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this

compatibility check.

Additionally, this routine will verify that the input matrix A is allocated with

appropriate upper bandwidth storage for the LU factorization.

Deprecated Name For backward compatibility, the wrapper function SUNBandLinearSolver with

idential input and output arguments is also provided.

F2003 Name FSUNLinSol\_Band

The SUNLINSOL\_BAND module defines band implementations of all "direct" linear solver operations listed in Sections 9.1.1 - 9.1.3:

- SUNLinSolGetType\_Band
- SUNLinSolInitialize\_Band this does nothing, since all consistency checks are performed at solver creation.

- SUNLinSolSetup\_Band this performs the LU factorization.
- ullet SUNLinSolSolve\_Band this uses the LU factors and pivots array to perform the solve.
- SUNLinSolLastFlag\_Band
- SUNLinSolSpace\_Band this only returns information for the storage within the solver object, i.e. storage for N, last\_flag, and pivots.
- SUNLinSolFree\_Band

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an 'F' to the function name.

#### 9.6.3 SUNLinearSolver\_Band Fortran interfaces

The SUNLINSOL\_BAND module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

#### FORTRAN 2003 interface module

The fsunlinsol\_band\_mod FORTRAN module defines interfaces to all SUNLINSOL\_BAND C functions using the intrinsic iso\_c\_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function SUNLinSol\_Band is interfaced as FSUNLinSol\_Band.

The FORTRAN 2003 SUNLINSOL\_BAND interface module can be accessed with the use statement, i.e. use fsunlinsol\_band\_mod, and linking to the library libsundials\_fsunlinsolband\_mod.lib in addition to the C library. For details on where the library and module file fsunlinsol\_band\_mod.mod are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the libsundials\_fsunlinsolband\_mod library.

#### FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL\_BAND module also includes a Fortran-callable function for creating a SUNLinearSolver object.

#### FSUNBANDLINSOLINIT

Call FSUNBANDLINSOLINIT(code, ier)

Description The function FSUNBANDLINSOLINIT can be called for Fortran programs to create a band SUNLinearSolver object.

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

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

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

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL\_BAND module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

#### FSUNMASSBANDLINSOLINIT

Call FSUNMASSBANDLINSOLINIT(ier)

Description The function FSUNMASSBANDLINSOLINIT can be called for Fortran programs to create a

band SUNLinearSolver object for mass matrix linear systems.

Arguments None

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes This routine must be called after both the NVECTOR and SUNMATRIX mass-matrix

objects have been initialized.

## 9.6.4 SUNLinearSolver\_Band content

The SUNLINSOL\_BAND module defines the *content* field of a SUNLinearSolver as the following structure:

```
struct _SUNLinearSolverContent_Band {
   sunindextype N;
   sunindextype *pivots;
   sunindextype last_flag;
};
```

These entries of the *content* field contain the following information:

N - size of the linear system,

pivots - index array for partial pivoting in LU factorization,

last\_flag - last error return flag from internal function evaluations.

# 9.7 The SUNLinearSolver\_LapackDense implementation

This section describes the SUNLINSOL implementation for solving dense linear systems with LA-PACK. The SUNLINSOL\_LAPACKDENSE module is designed to be used with the corresponding SUNMA-TRIX\_DENSE matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR\_SERIAL, NVECTOR\_OPENMP, or NVECTOR\_PTHREADS).

To access the SUNLINSOL\_LAPACKDENSE module, include the header file sunlinsol/sunlinsol\_lapackdense.h. The installed module library to link to is libsundials\_sunlinsollapackdense.lib where .lib is typically .so for shared libraries and .a for static libraries.

The SUNLINSOL\_LAPACKDENSE module is a SUNLINSOL wrapper for the LAPACK dense matrix factorization and solve routines, \*GETRF and \*GETRS, where \* is either D or S, depending on whether SUNDIALS was configured to have realtype set to double or single, respectively (see Section 4.2). In order to use the SUNLINSOL\_LAPACKDENSE module it is assumed that LAPACK has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with LAPACK (see Appendix A for details). We note that since there do not exist 128-bit floating-point factorization and solve routines in LAPACK, this interface cannot be compiled when using extended precision for realtype. Similarly, since there do not exist 64-bit integer LAPACK routines, the SUNLINSOL\_LAPACKDENSE module also cannot be compiled when using 64-bit integers for the sunindextype.



# 9.7.1 SUNLinearSolver\_LapackDense description

This solver is constructed to perform the following operations:

- The "setup" call performs a LU factorization with partial (row) pivoting  $(\mathcal{O}(N^3) \cos t)$ , 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  $(\mathcal{O}(N^2) \text{ cost})$ .

# 9.7.2 SUNLinearSolver\_LapackDense functions

The SUNLINSOL\_LAPACKDENSE module provides the following user-callable constructor for creating a SUNLinearSolver object.

# SUNLinSol\_LapackDense

Call LS = SUNLinSol\_LapackDense(y, A);

Description The function SUNLinSol\_LapackDense creates and allocates memory for a LAPACK-

based, dense SUNLinearSolver object.

Arguments y (N\_Vector) a template for cloning vectors needed within the solver

A (SUNMatrix) a  ${\tt SUNMATRIX\_DENSE}$  matrix template for cloning matrices needed

within the solver

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

routine will return NULL.

Notes This routine will perform consistency checks to ensure that it is called with con-

sistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX\_DENSE matrix type and the NVECTOR\_SERIAL, NVECTOR\_OPENMP, and NVECTOR\_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this

compatibility check.

 $\label{thm:local_problem} Deprecated \ Name \ For \ backward \ compatibility, \ the \ wrapper \ function \ {\tt SUNLapackDense} \ with \ idential$ 

input and output arguments is also provided.

The SUNLINSOL\_LAPACKDENSE module defines dense implementations of all "direct" linear solver operations listed in Sections 9.1.1 - 9.1.3:

- SUNLinSolGetType\_LapackDense
- SUNLinSolInitialize\_LapackDense this does nothing, since all consistency checks are performed at solver creation.
- SUNLinSolSetup\_LapackDense this calls either DGETRF or SGETRF to perform the LU factorization.
- SUNLinSolSolve\_LapackDense this calls either DGETRS or SGETRS to use the *LU* factors and pivots array to perform the solve.
- $\bullet \ {\tt SUNLinSolLastFlag\_LapackDense} \\$
- SUNLinSolSpace\_LapackDense this only returns information for the storage within the solver object, i.e. storage for N, last\_flag, and pivots.
- SUNLinSolFree\_LapackDense

# 9.7.3 SUNLinearSolver\_LapackDense Fortran interfaces

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL\_LAPACKDENSE module also includes a Fortran-callable function for creating a SUNLinearSolver object.

## FSUNLAPACKDENSEINIT

Call FSUNLAPACKDENSEINIT(code, ier)

Description The function FSUNLAPACKDENSEINIT can be called for Fortran programs to create a

LAPACK-based dense SUNLinearSolver object.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for

KINSOL, and 4 for ARKODE).

Return value ier is a return completion flag equal to 0 for a success return and -1 otherwise. See

printed message for details in case of failure.

Notes This routine must be called *after* both the NVECTOR and SUNMATRIX objects have been

initialized.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL\_LAPACKDENSE module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

#### FSUNMASSLAPACKDENSEINIT

Call FSUNMASSLAPACKDENSEINIT(ier)

Description The function FSUNMASSLAPACKDENSEINIT can be called for Fortran programs to create

a LAPACK-based, dense SUNLinearSolver object for mass matrix linear systems.

Arguments None

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes This routine must be called after both the NVECTOR and SUNMATRIX mass-matrix

objects have been initialized.

# 9.7.4 SUNLinearSolver\_LapackDense content

The SUNLINSOL\_LAPACKDENSE module defines the *content* field of a SUNLinearSolver as the following structure:

```
struct _SUNLinearSolverContent_Dense {
   sunindextype N;
   sunindextype *pivots;
   sunindextype last_flag;
};
```

These entries of the *content* field contain the following information:

N - size of the linear system,

pivots - index array for partial pivoting in LU factorization,

last\_flag - last error return flag from internal function evaluations.

# 9.8 The SUNLinearSolver\_LapackBand implementation

This section describes the SUNLINSOL implementation for solving banded linear systems with LA-PACK. The SUNLINSOL\_LAPACKBAND module is designed to be used with the corresponding SUNMA-TRIX\_BAND matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR\_SERIAL, NVECTOR\_OPENMP, or NVECTOR\_PTHREADS).

To access the SUNLINSOL\_LAPACKBAND module, include the header file sunlinsol/sunlinsol\_lapackband.h. The installed module library to link to is libsundials\_sunlinsollapackband.lib where .lib is typically .so for shared libraries and .a for static libraries.

The SUNLINSOL\_LAPACKBAND module is a SUNLINSOL wrapper for the LAPACK band matrix factorization and solve routines, \*GBTRF and \*GBTRS, where \* is either D or S, depending on whether SUNDIALS was configured to have realtype set to double or single, respectively (see Section 4.2). In order to use the SUNLINSOL\_LAPACKBAND module it is assumed that LAPACK has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with LAPACK (see Appendix A for details). We note that since there do not exist 128-bit floating-point factorization and solve routines in LAPACK, this interface cannot be compiled when using extended precision for realtype. Similarly, since there do not exist 64-bit integer LAPACK routines, the SUNLINSOL\_LAPACKBAND module also cannot be compiled when using 64-bit integers for the sunindextype.



#### SUNLinearSolver\_LapackBand description 9.8.1

This solver is constructed to perform the following operations:

- The "setup" call performs a LU factorization with partial (row) pivoting, PA = LU, where P is a permutation matrix, L is a lower triangular matrix with 1's on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX\_BAND object A, with pivoting information encoding P stored in the pivots array.
- The "solve" call performs pivoting and forward and backward substitution using the stored pivots array and the LU factors held in the SUNMATRIX\_BAND object.
- A must be allocated to accommodate the increase in upper bandwidth that occurs during factorization. More precisely, if A is a band matrix with upper bandwidth mu and lower bandwidth ml, then the upper triangular factor U can have upper bandwidth as big as smu = MIN(N-1, mu+ml). The lower triangular factor L has lower bandwidth m1.



#### 9.8.2SUNLinearSolver\_LapackBand functions

The SUNLINSOL\_LAPACKBAND module provides the following user-callable constructor for creating a SUNLinearSolver object.

SUNLinSol_LapackBand
DONDINDOT_DapachDane

Call LS = SUNLinSol\_LapackBand(y, A);

Description The function SUNLinSol\_LapackBand creates and allocates memory for a LAPACK-

based, band SUNLinearSolver object.

Arguments y (N\_Vector) a template for cloning vectors needed within the solver

A (SUNMatrix) a SUNMATRIX\_BAND matrix template for cloning matrices needed

within the solver

Return value This returns a SUNLinear Solver 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 con-

sistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX\_BAND matrix type and the NVECTOR\_SERIAL, NVECTOR\_OPENMP, and NVECTOR\_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this

compatibility check.

Additionally, this routine will verify that the input matrix A is allocated with

appropriate upper bandwidth storage for the LU factorization.

Deprecated Name For backward compatibility, the wrapper function SUNLapackBand with idential input and output arguments is also provided.

The SUNLINSOL\_LAPACKBAND module defines band implementations of all "direct" linear solver operations listed in Sections 9.1.1 - 9.1.3:

- SUNLinSolGetType\_LapackBand
- SUNLinSolInitialize\_LapackBand this does nothing, since all consistency checks are performed at solver creation.
- SUNLinSolSetup\_LapackBand this calls either DGBTRF or SGBTRF to perform the LU factorization.
- SUNLinSolSolve\_LapackBand this calls either DGBTRS or SGBTRS to use the *LU* factors and pivots array to perform the solve.
- SUNLinSolLastFlag\_LapackBand
- SUNLinSolSpace\_LapackBand this only returns information for the storage within the solver object, i.e. storage for N, last\_flag, and pivots.
- SUNLinSolFree\_LapackBand

# 9.8.3 SUNLinearSolver\_LapackBand Fortran interfaces

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL\_LAPACKBAND module also includes a Fortran-callable function for creating a SUNLinearSolver object.

# FSUNLAPACKDENSEINIT

Call FSUNLAPACKBANDINIT(code, ier)

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

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

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

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

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL\_LAPACKBAND module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

#### FSUNMASSLAPACKBANDINIT

Call FSUNMASSLAPACKBANDINIT(ier)

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

Arguments None

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

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

# 9.8.4 SUNLinearSolver\_LapackBand content

The SUNLINSOL\_LAPACKBAND module defines the *content* field of a SUNLinearSolver as the following structure:

# 9.9 The SUNLinearSolver\_KLU implementation

This section describes the SUNLINSOL implementation for solving sparse linear systems with KLU. The SUNLINSOL\_KLU module is designed to be used with the corresponding SUNMATRIX\_SPARSE matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR\_SERIAL, NVECTOR\_OPENMP, or NVECTOR\_PTHREADS).

The header file to include when using this module is sunlinsol/sunlinsol\_klu.h. The installed module library to link to is libsundials\_sunlinsolklu.lib where .lib is typically .so for shared libraries and .a for static libraries.

The SUNLINSOL\_KLU module is a SUNLINSOL wrapper for the KLU sparse matrix factorization and solver library written by Tim Davis [3, 22]. In order to use the SUNLINSOL\_KLU interface to KLU, it is assumed that KLU has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with KLU (see Appendix A for details). Additionally, this wrapper only supports double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to have realtype set to either extended or single (see Section 4.2). Since the KLU library supports both 32-bit and 64-bit integers, this interface will be compiled for either of the available sunindextype options.

# <u>!</u>

# 9.9.1 SUNLinearSolver\_KLU description

The KLU library has a symbolic factorization routine that computes the permutation of the linear system matrix to block triangular form and the permutations that will pre-order the diagonal blocks (the only ones that need to be factored) to reduce fill-in (using AMD, COLAMD, CHOLAMD, natural, or an ordering given by the user). Of these ordering choices, the default value in the SUNLINSOL\_KLU module is the COLAMD ordering.

KLU breaks the factorization into two separate parts. The first is a symbolic factorization and the second is a numeric factorization that returns the factored matrix along with final pivot information. KLU also has a refactor routine that can be called instead of the numeric factorization. This routine will reuse the pivot information. This routine also returns diagnostic information that a user can examine to determine if numerical stability is being lost and a full numerical factorization should be done instead of the refactor.

Since the linear systems that arise within the context of SUNDIALS calculations will typically have identical sparsity patterns, the SUNLINSOL\_KLU module is constructed to perform the following operations:

• The first time that the "setup" routine is called, it performs the symbolic factorization, followed by an initial numerical factorization.

- On subsequent calls to the "setup" routine, it calls the appropriate KLU "refactor" routine, followed by estimates of the numerical conditioning using the relevant "rcond", and if necessary "condest", routine(s). If these estimates of the condition number are larger than  $\varepsilon^{-2/3}$  (where  $\varepsilon$  is the double-precision unit roundoff), then a new factorization is performed.
- The module includes the routine SUNKLUReInit, that can be called by the user to force a full or partial refactorization at the next "setup" call.
- The "solve" call performs pivoting and forward and backward substitution using the stored KLU data structures. We note that in this solve KLU operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

## 9.9.2 SUNLinearSolver\_KLU functions

The SUNLINSOL\_KLU module provides the following user-callable constructor for creating a SUNLinearSolver object.

#### SUNLinSol\_KLU

Call LS = SUNLinSol\_KLU(y, A);

Description The function SUNLinSol\_KLU creates and allocates memory for a KLU-based

SUNLinearSolver object.

Arguments y (N\_Vector) a template for cloning vectors needed within the solver

A (SUNMatrix) a SUNMATRIX\_SPARSE matrix template for cloning matrices needed

within the solver

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

routine will return NULL.

Notes This routine will perform consistency checks to ensure that it is called with con-

sistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX\_SPARSE matrix type (using either CSR or CSC storage formats) and the NVECTOR\_SERIAL, NVECTOR\_OPENMP, and NVECTOR\_PTHREADS vector types. As additional compatible matrix and vector implementations are added to

SUNDIALS, these will be included within this compatibility check.

Deprecated Name For backward compatibility, the wrapper function SUNKLU with idential input and

output arguments is also provided.

F2003 Name FSUNLinSol\_KLU

The SUNLINSOL\_KLU module defines implementations of all "direct" linear solver operations listed in Sections 9.1.1 - 9.1.3:

- SUNLinSolGetType\_KLU
- SUNLinSolInitialize\_KLU this sets the first\_factorize flag to 1, forcing both symbolic and numerical factorizations on the subsequent "setup" call.
- ullet SUNLinSolSetup\_KLU this performs either a LU factorization or refactorization of the input matrix.
- SUNLinSolSolve\_KLU this calls the appropriate KLU solve routine to utilize the *LU* factors to solve the linear system.
- SUNLinSolLastFlag\_KLU
- SUNLinSolSpace\_KLU this only returns information for the storage within the solver *interface*, i.e. storage for the integers last\_flag and first\_factorize. For additional space requirements, see the KLU documentation.

#### • SUNLinSolFree\_KLU

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an 'F' to the function name.

The SUNLINSOL\_KLU module also defines the following additional user-callable functions.

#### SUNLinSol\_KLUReInit

Call retval = SUNLinSol\_KLUReInit(LS, A, nnz, reinit\_type);

Description The function SUNLinSol\_KLUReInit reinitializes memory and flags for a new fac-

torization (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 LS (SUNLinearSolver) a template for cloning vectors needed within the

solver

A (SUNMatrix) a SUNMATRIX\_SPARSE matrix template for cloning ma-

trices needed within the solver

nnz (sunindextype) the new number of nonzeros in the matrix

reinit\_type (int) flag governing the level of reinitialization. The allowed values
 are:

- SUNKLU\_REINIT\_FULL The Jacobian matrix will be destroyed and a new one will be allocated based on the nnz value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup.
- SUNKLU\_REINIT\_PARTIAL Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of nnz given in the sparse matrix provided to the original constructor routine (or the previous SUNLinSol\_KLUReInit call).

Return value The return values from this function are SUNLS\_MEM\_NULL (either S or A are NULL),

SUNLS\_ILL\_INPUT (A does not have type SUNMATRIX\_SPARSE or reinit\_type is invalid), SUNLS\_MEM\_FAIL (reallocation of the sparse matrix failed) or SUNLS\_SUCCESS.

Notes This routine will perform consistency checks to ensure that it is called with con-

sistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX\_SPARSE matrix type (using either CSR or CSC storage formats) and the NVECTOR\_SERIAL, NVECTOR\_OPENMP, and NVECTOR\_PTHREADS vector types. As additional compatible matrix and vector implementations are added to

SUNDIALS, these will be included within this compatibility check.

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

Deprecated Name For backward compatibility, the wrapper function SUNKLUReInit with idential in-

put and output arguments is also provided.

F2003 Name FSUNLinSol\_KLUReInit

# SUNLinSol\_KLUSetOrdering

Call retval = SUNLinSol\_KLUSetOrdering(LS, ordering);

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

Arguments LS (SUNLinearSolver) the SUNLINSOL\_KLU object

ordering (int) flag indicating the reordering algorithm to use, the options are:

0 AMD,

1 COLAMD, and 2 the natural ordering.

The default is 1 for COLAMD.

Return value The return values from this function are SUNLS\_MEM\_NULL (S is NULL),

SUNLS\_ILL\_INPUT (invalid ordering choice), or SUNLS\_SUCCESS.

Deprecated Name For backward compatibility, the wrapper function SUNKLUSetOrdering with iden-

tial input and output arguments is also provided.

F2003 Name FSUNLinSol\_KLUSetOrdering

# SUNLinSol\_KLUGetSymbolic

Call symbolic = SUNLinSol\_KLUGetSymbolic(LS);

Description This function returns a pointer to the KLU symbolic factorization stored in the SUNLIN-

SOL\_KLU content structure.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_KLU object

Return value The return type from this function is sun\_klu\_symbolic.

Notes When SUNDIALS is compiled with 32-bit indices (SUNDIALS\_INDEX\_SIZE=32),

sun\_klu\_symbolic is mapped to the KLU type klu\_symbolic; when SUNDIALS is compiled with 64-bit indices (SUNDIALS\_INDEX\_SIZE=64) this is mapped to the KLU type

klu\_l\_symbolic.

#### SUNLinSol\_KLUGetNumeric

Call numeric = SUNLinSol\_KLUGetNumeric(LS);

Description This function returns a pointer to the KLU numeric factorization stored in the SUNLIN-

SOL\_KLU content structure.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_KLU object

Return value The return type from this function is sun\_klu\_numeric.

Notes When SUNDIALS is compiled with 32-bit indices (SUNDIALS\_INDEX\_SIZE=32),

sun\_klu\_numeric is mapped to the KLU type klu\_numeric; when SUNDIALS is compiled with 64-bit indices (SUNDIALS\_INDEX\_SIZE=64), this is mapped to the KLU type

klu\_l\_numeric.

#### SUNLinSol\_KLUGetCommon

Description This function returns a pointer to the KLU common structure stored within in the

SUNLINSOL\_KLU content structure.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_KLU object

Return value The return type from this function is sun\_klu\_common.

Notes 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

 $klu\_l\_common.$ 

# 9.9.3 SUNLinearSolver\_KLU Fortran interfaces

The SUNLINSOL\_KLU module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

#### FORTRAN 2003 interface module

The fsunlinsol\_klu\_mod FORTRAN module defines interfaces to all SUNLINSOL\_KLU C functions using the intrinsic iso\_c\_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function SUNLinSol\_klu is interfaced as FSUNLinSol\_klu.

The FORTRAN 2003 SUNLINSOL\_KLU interface module can be accessed with the use statement, i.e. use fsunlinsol\_klu\_mod, and linking to the library libsundials\_fsunlinsolklu\_mod.lib in addition to the C library. For details on where the library and module file fsunlinsol\_klu\_mod.mod are installed see Appendix A.

#### FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL\_KLU module also includes a Fortran-callable function for creating a SUNLinearSolver object.

#### FSUNKLUINIT

Call FSUNKLUINIT(code, ier)

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

SOL\_KLU object.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for

KINSOL, and 4 for ARKODE).

Return value  $\mathtt{ier}$  is a return completion flag equal to 0 for a success return and  $\mathtt{-1}$  otherwise. See

printed message for details in case of failure.

Notes This routine must be called *after* both the NVECTOR and SUNMATRIX objects have been

initialized.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL\_KLU module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

# FSUNMASSKLUINIT

Call FSUNMASSKLUINIT(ier)

Description The function FSUNMASSKLUINIT can be called for Fortran programs to create a KLU-

based SUNLinearSolver object for mass matrix linear systems.

Arguments None

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes This routine must be called after both the NVECTOR and SUNMATRIX mass-matrix

objects have been initialized.

The SUNLinSol\_KLUReInit and SUNLinSol\_KLUSetOrdering routines also support FORTRAN interfaces for the system and mass matrix solvers:

#### FSUNKLUREINIT

Call FSUNKLUREINIT(code, nnz, reinit\_type, ier)

Description The function FSUNKLUREINIT can be called for Fortran programs to re-initialize a SUN-

LINSOL\_KLU object.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA,

3 for KINSOL, and 4 for ARKODE).

nnz (sunindextype\*) the new number of nonzeros in the matrix

reinit\_type (int\*) flag governing the level of reinitialization. The allowed values are:

- 1 The Jacobian matrix will be destroyed and a new one will be allocated based on the nnz value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup.
- 2 Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of nnz given in the sparse matrix provided to the original constructor routine (or the previous SUNLinSol\_KLUReInit call).

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

Notes See SUNLinSol\_KLUReInit for complete further documentation of this routine.

## FSUNMASSKLUREINIT

Call FSUNMASSKLUREINIT(nnz, reinit\_type, ier)

Description The function FSUNMASSKLUREINIT can be called for Fortran programs to re-initialize a

SUNLINSOL\_KLU object for mass matrix linear systems.

Arguments The arguments are identical to FSUNKLUREINIT above, except that code is not needed

since mass matrix linear systems only arise in ARKODE.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_KLUReInit for complete further documentation of this routine.

#### FSUNKLUSETORDERING

Call FSUNKLUSETORDERING(code, ordering, ier)

reordering algorithm used by KLU.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3

for KINSOL, and 4 for ARKODE).

ordering (int\*) flag indication the reordering algorithm to use. Options include:

0 AMD,

1 COLAMD, and

2 the natural ordering.

The default is 1 for COLAMD.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_KLUSetOrdering for complete further documentation of this routine.

# FSUNMASSKLUSETORDERING

Call FSUNMASSKLUSETORDERING(ier)

Description The function FSUNMASSKLUSETORDERING can be called for Fortran programs to change

the reordering algorithm used by KLU for mass matrix linear systems.

Arguments The arguments are identical to FSUNKLUSETORDERING above, except that code is not

needed since mass matrix linear systems only arise in ARKODE.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_KLUSetOrdering for complete further documentation of this routine.

#### 9.9.4 SUNLinearSolver\_KLU content

The SUNLINSOL\_KLU module defines the *content* field of a SUNLinearSolver as the following structure:

```
struct _SUNLinearSolverContent_KLU {
  int
                     last_flag;
                     first_factorize;
  int
  sun_klu_symbolic *symbolic;
  sun_klu_numeric
                     *numeric;
  sun_klu_common
                     common;
                      (*klu_solver)(sun_klu_symbolic*, sun_klu_numeric*,
  sunindextype
                                      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,
                 - KLU storage structure for numeric factorization components, with underlying type
numeric
                 klu_numeric or klu_l_numeric, depending on whether SUNDIALS was installed with
                 32-bit versus 64-bit indices, respectively.
                  - storage structure for common KLU solver components, with underlying type
common
                 klu_common or klu_l_common, depending on whether SUNDIALS was installed with
                 32-bit versus 64-bit indices, respectively,
                 - pointer to the appropriate KLU solver function (depending on whether it is using
klu_solver
                 a CSR or CSC sparse matrix, and on whether SUNDIALS was installed with 32-bit
                 or 64-bit indices).
```

# 9.10 The SUNLinearSolver\_SuperLUDIST implementation

The SuperLU\_DIST implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL\_SUPERLUDIST, is designed to be used with the corresponding SUNMATRIX\_SLUNRLOC matrix type, and one of the serial, threaded or parallel NVECTOR implementations (NVECTOR\_SERIAL, NVECTOR\_PTHREADS, NVECTOR\_PARALLEL, or NVECTOR\_PARHYP).

The header file to include when using this module is sunlinsol/sunlinsol\_superludist.h. The installed module library to link to is libsundials\_sunlinsolsuperludist.lib where .lib is typically .so for shared libraries and .a for static libraries.

## 9.10.1 SUNLinearSolver\_SuperLUDIST description

The Sunlinsol\_Superludist module is a sunlinsol adapter for the Superlu\_DIST sparse matrix factorization and solver library written by X. Sherry Li [8, 28, 40, 41]. The package uses a SPMD parallel programming model and multithreading to enhance efficiency in distributed-memory parallel environments with multicore nodes and possibly GPU accelerators. It uses MPI for communication, OpenMP for threading, and CUDA for GPU support. In order to use the SUNLINSOL\_SUPERLUDIST interface to Superlu\_DIST, it is assumed that Superlu\_DIST has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with Superlu\_DIST (see Appendix A for details). Additionally, the adapter only supports double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to use single or extended precision. Moreover, since the Superlu\_DIST library may be installed to support either 32-bit or

64-bit integers, it is assumed that the SuperLU\_DIST library is installed using the same integer size as SUNDIALS.

The SuperLU\_DIST library provides many options to control how a linear system will be solved. These options may be set by a user on an instance of the superlu\_dist\_options\_t struct, and then it may be provided as an argument to the SUNLINSOL\_SUPERLUDIST constructor. The SUNLINSOL\_SUPERLUDIST module will respect all options set except for Fact – this option is necessarily modified by the SUNLINSOL\_SUPERLUDIST module in the setup and solve routines.

Since the linear systems that arise within the context of SUNDIALS calculations will typically have identical sparsity patterns, the SUNLINSOL\_SUPERLUDIST module is constructed to perform the following operations:

- The first time that the "setup" routine is called, it sets the SuperLU\_DIST option Fact to DOFACT so that a subsequent call to the "solve" routine will perform a symbolic factorization, followed by an initial numerical factorization before continuing to solve the system.
- On subsequent calls to the "setup" routine, it sets the SuperLU\_DIST option Fact to SamePattern so that a subsequent call to "solve" will perform factorization assuming the same sparsity pattern as prior, i.e. it will reuse the column permutation vector.
- If "setup" is called prior to the "solve" routine, then the "solve" routine will perform a symbolic factorization, followed by an initial numerical factorization before continuing to the sparse triangular solves, and, potentially, iterative refinement. If "setup" is not called prior, "solve" will skip to the triangular solve step. We note that in this solve SuperLU\_DIST operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.



Starting with SuperLU\_DIST version 6.3.0, some structures were renamed to have a prefix for the floating point type. The double precision API functions have the prefix 'd'. To maintain backwards compatibility with the 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.

# 9.10.2 SUNLinearSolver\_SuperLUDIST functions

The SUNLINSOL\_SUPERLUDIST module defines implementations of all "direct" linear solver operations listed in Sections 9.1.1-9.1.3:

- SUNLinSolGetType\_SuperLUDIST
- SUNLinSolInitialize\_SuperLUDIST this sets the first\_factorize flag to 1 and resets the internal SuperLU\_DIST statistics variables.
- SUNLinSolSetup\_SuperLUDIST this sets the appropriate SuperLU\_DIST options so that a subsequent solve will perform a symbolic and numerical factorization before proceeding with the triangular solves
- SUNLinSolsolve\_SuperLUDIST this calls the SuperLU\_DIST solve routine to perform factorization (if the setup routine was called prior) and then use the *LU* factors to solve the linear system.
- SUNLinSolLastFlag\_SuperLUDIST
- SUNLinSolSpace\_SuperLUDIST this only returns information for the storage within the solver *interface*, i.e. storage for the integers last\_flag and first\_factorize. For additional space requirements, see the SuperLU\_DIST documentation.
- SUNLinSolFree\_SuperLUDIST

In addition, the module SUNLINSOL\_SUPERLUDIST provides the following user-callable routines:

#### SUNLinSol\_SuperLUDIST

Call LS = SUNLinSol\_SuperLUDIST(y, A, grid, lu, scaleperm, solve, stat, options);

Description The function SUNLinSol\_SuperLUDIST creates and allocates memory for a SUNLIN-

SOL\_SUPERLUDIST object.

Arguments y (N\_Vector) a template for cloning vectors needed within the solver

A (SUNMatrix) a SUNMATRIX\_SLUNRLOC matrix template for cloning matrices

needed within the solver

grid (gridinfo\_t\*)
lu (LUstruct\_t\*)

scaleperm (ScalePermstruct\_t\*)

solve (SOLVEstruct\_t\*)
stat (SuperLUStat\_t\*)

options (superlu\_dist\_options\_t\*)

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

Notes This routine analyzes the input matrix and vector to determine the linear system size and to assess compatibility with the SuperLU\_DIST library.

This routine will perform consistency checks to ensure that it is called with consistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX\_SLUNRLOC matrix type and the NVECTOR\_SERIAL, NVECTOR\_PARALLEL, NVECTOR\_PARHYP, NVECTOR\_OPENMP, and NVECTOR\_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

The grid, lu, scaleperm, solve, and options arguments are not checked and are passed directly to SuperLU\_DIST routines.

Some struct members of the options argument are modified internally by the SUNLIN-SOL\_SUPERLUDIST solver. Specifically the member Fact, is modified in the setup and solve routines.

#### SUNLinSol\_SuperLUDIST\_GetBerr

Call realtype berr = SUNLinSol\_SuperLUDIST\_GetBerr(LS);

Description The function SUNLinSol\_SuperLUDIST\_GetBerr returns the componentwise relative back-

ward error of the computed solution.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SUPERLUDIST object

Return value realtype

Notes

#### |SUNLinSol\_SuperLUDIST\_GetGridinfo

Call gridinfo\_t \*grid = SUNLinSol\_SuperLUDIST\_GetGridinfo(LS);

Description The function SUNLinSol\_SuperLUDIST\_GetGridinfo returns the SuperLU\_DIST struc-

ture that contains the 2D process grid.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SUPERLUDIST object

Return value gridinfo\_t\*

Notes

#### SUNLinSol\_SuperLUDIST\_GetLUstruct

Call LUstruct\_t \*lu = SUNLinSol\_SuperLUDIST\_GetLUstruct(LS);

Description The function SUNLinSol\_SuperLUDIST\_GetLUstruct returns the SuperLU\_DIST struc-

ture that contains the distributed L and U factors.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SUPERLUDIST object

Return value LUstruct\_t\*

Notes

#### SUNLinSol\_SuperLUDIST\_GetSuperLUOptions

Call superlu\_dist\_options\_t \*opts = SUNLinSol\_SuperLUDIST\_GetSuperLUOptions(LS);

Description The function SUNLinSol\_SuperLUDIST\_GetSuperLUOptions returns the SuperLU\_DIST

structure that contains the options which control how the linear system is factorized

and solved.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SUPERLUDIST object

Return value superlu\_dist\_options\_t\*

Notes

#### ${\tt SUNLinSol\_SuperLUDIST\_GetScalePermstruct}$

Call ScalePermstruct\_t \*sp = SUNLinSol\_SuperLUDIST\_GetScalePermstruct(LS);

Description The function SUNLinSol\_SuperLUDIST\_GetScalePermstruct returns the SuperLU\_DIST

structure that contains the vectors that describe the transformations done to the matrix,

A.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SUPERLUDIST object

Return value ScalePermstruct\_t\*

Notes

#### |SUNLinSol\_SuperLUDIST\_GetSOLVEstruct

Call SOLVEstruct\_t \*solve = SUNLinSol\_SuperLUDIST\_GetSOLVEstruct(LS);

Description The function SUNLinSol\_SuperLUDIST\_GetSOLVEstruct returns the SuperLU\_DIST struc-

ture that contains information for communication during the solution phase.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SUPERLUDIST object

Return value SOLVEstruct\_t\*

Notes

#### ${\tt SUNLinSol\_SuperLUDIST\_GetSuperLUStat}$

Call SuperLUStat\_t \*stat = SUNLinSol\_SuperLUDIST\_GetSuperLUStat(LS);

Description The function SUNLinSol\_SuperLUDIST\_GetSuperLUStat returns the SuperLU\_DIST struc-

ture that stores information about runtime and flop count.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SUPERLUDIST object

Return value SuperLUStat\_t\*

Notes

#### 9.10.3 SUNLinearSolver\_SuperLUDIST content

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

```
struct _SUNLinearSolverContent_SuperLUDIST {
  booleantype
                           first_factorize;
  int
                           last_flag;
  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 calls to internal routines,

berr - the componentwise relative backward error of the computed solution,

grid - pointer to the SuperLU\_DIST structure that stores the 2D process grid,

 $\mathbf{lu}$  - pointer to the SuperLU\_DIST structure that stores the distributed L and U factors,

options - pointer to SuperLU\_DIST options structure,

 ${\bf scaleperm}$  - pointer to the SuperLU\_DIST structure that stores vectors describing the transformations done to the matrix, A,

solve - pointer to the SuperLU\_DIST solve structure,

stat - pointer to the SuperLU\_DIST structure that stores information about runtime and flop count,

N - the number of equations in the system

## 9.11 The SUNLinearSolver\_SuperLUMT implementation

This section describes the SUNLINSOL implementation for solving sparse linear systems with SuperLU\_MT. The SUPERLUMT module is designed to be used with the corresponding SUNMATRIX\_SPARSE matrix type, and one of the serial or shared-memory NVECTOR implementations (NVECTOR\_SERIAL, NVECTOR\_OPENMP, or NVECTOR\_PTHREADS). While these are compatible, it is not recommended to use a threaded vector module with SUNLINSOL\_SUPERLUMT unless it is the NVECTOR\_OPENMP module and the SUPERLUMT library has also been compiled with OpenMP.

The header file to include when using this module is sunlinsol/sunlinsol\_superlumt.h. The installed module library to link to is libsundials\_sunlinsolsuperlumt.lib where .lib is typically .so for shared libraries and .a for static libraries.

The SUNLINSOL\_SUPERLUMT module is a SUNLINSOL wrapper for the SUPERLUMT sparse matrix factorization and solver library written by X. Sherry Li [9, 39, 24]. The package performs matrix factorization using threads to enhance efficiency in shared memory parallel environments. It should be noted that threads are only used in the factorization step. In order to use the SUNLINSOL\_SUPERLUMT interface to SUPERLUMT, it is assumed that SUPERLUMT has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with SUPERLUMT



(see Appendix A for details). Additionally, this wrapper only supports single- and double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to have realtype set to extended (see Section 4.2). Moreover, since the SUPERLUMT library may be installed to support either 32-bit or 64-bit integers, it is assumed that the SUPERLUMT library is installed using the same integer precision as the SUNDIALS sunindextype option.

#### 9.11.1SUNLinearSolver\_SuperLUMT description

The SUPERLUMT library has a symbolic factorization routine that computes the permutation of the linear system matrix to reduce fill-in on subsequent LU factorizations (using COLAMD, minimal degree ordering on  $A^T * A$ , minimal degree ordering on  $A^T + A$ , or natural ordering). Of these ordering choices, the default value in the SUNLINSOL\_SUPERLUMT module is the COLAMD ordering.

Since the linear systems that arise within the context of SUNDIALS calculations will typically have identical sparsity patterns, the SUNLINSOL\_SUPERLUMT module is constructed to perform the following operations:

- The first time that the "setup" routine is called, it performs the symbolic factorization, followed by an initial numerical factorization.
- On subsequent calls to the "setup" routine, it skips the symbolic factorization, and only refactors the input matrix.
- The "solve" call performs pivoting and forward and backward substitution using the stored SUPERLUMT data structures. We note that in this solve SUPERLUMT operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

## SUNLinearSolver\_SuperLUMT functions

The module SUNLINSOL\_SUPERLUMT provides the following user-callable constructor for creating a SUNLinearSolver object.

SUNLinSol_Super	LUMT				
Call	LS = SUNLinSol_SuperLUMT(y, A, num_threads);				
Description	The function SUNLinSol_SuperLUMT creates and allocates memory for a SuperLU_MT-based SUNLinearSolver object.				
Arguments	y A	$({\tt N\_Vector})$ a template for cloning vectors needed within the solver $({\tt SUNMatrix})$ a ${\tt SUNMATRIX\_SPARSE}$ matrix template for cloning matrices needed within the solver			
	$num\_threads$	$({\tt int})$ desired number of threads (OpenMP or Pthreads, depending on how SUPERLUMT was installed) to use during the factorization steps			
Return value	This returns a SUNLinearSolver object. If either A or y are incompatible then this routine will return NULL.				
Notes	This routine analyzes the input matrix and vector to determine the linear system				

size and to assess compatibility with the SUPERLUMT library.

This routine will perform consistency checks to ensure that it is called with consistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX\_SPARSE matrix type (using either CSR or CSC storage formats) and the NVECTOR\_SERIAL, NVECTOR\_OPENMP, and NVECTOR\_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

The num\_threads argument is not checked and is passed directly to SUPERLUMT routines.

Deprecated Name For backward compatibility, the wrapper function SUNSuperLUMT with idential input and output arguments is also provided.

The SUNLINSOL\_SUPERLUMT module defines implementations of all "direct" linear solver operations listed in Sections 9.1.1 - 9.1.3:

- SUNLinSolGetType\_SuperLUMT
- SUNLinSolInitialize\_SuperLUMT this sets the first\_factorize flag to 1 and resets the internal SUPERLUMT statistics variables.
- SUNLinSolSetup\_SuperLUMT this performs either a *LU* factorization or refactorization of the input matrix.
- SUNLinSolSolve\_SuperLUMT this calls the appropriate SUPERLUMT solve routine to utilize the *LU* factors to solve the linear system.
- SUNLinSolLastFlag\_SuperLUMT
- SUNLinSolSpace\_SuperLUMT this only returns information for the storage within the solver *interface*, i.e. storage for the integers last\_flag and first\_factorize. For additional space requirements, see the SUPERLUMT documentation.
- SUNLinSolFree\_SuperLUMT

The SUNLINSOL\_SUPERLUMT module also defines the following additional user-callable function.

#### SUNLinSol\_SuperLUMTSetOrdering

Call retval = SUNLinSol\_SuperLUMTSetOrdering(LS, ordering);

Description This function sets the ordering used by SUPERLUMT for reducing fill in the linear

solve.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SUPERLUMT object

ordering (int) a flag indicating the ordering algorithm to use, the options are:

0 natural ordering

1 minimal degree ordering on  $A^TA$ 

2 minimal degree ordering on  $A^T + A$ 

3 COLAMD ordering for unsymmetric matrices

The default is 3 for COLAMD.

Return value The return values from this function are SUNLS\_MEM\_NULL (S is NULL),

SUNLS\_ILL\_INPUT (invalid ordering choice), or SUNLS\_SUCCESS.

Deprecated Name For backward compatibility, the wrapper function SUNSuperLUMTSetOrdering with idential input and output arguments is also provided.

#### 9.11.3 SUNLinearSolver\_SuperLUMT Fortran interfaces

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

#### FSUNSUPERLUMTINIT

Call FSUNSUPERLUMTINIT(code, num\_threads, ier)

Description The function FSUNSUPERLUMTINIT can be called for Fortran programs to create a sun-

LINSOL\_KLU object.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA,

3 for KINSOL, and 4 for ARKODE).

num\_threads (int\*) desired number of threads (OpenMP or Pthreads, depending on how SUPERLUMT was installed) to use during the factorization steps

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

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

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL\_SUPERLUMT module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

#### FSUNMASSSUPERLUMTINIT

Call FSUNMASSSUPERLUMTINIT(num\_threads, ier)

Description The function FSUNMASSSUPERLUMTINIT can be called for Fortran programs to create a SuperLU\_MT-based SUNLinearSolver object for mass matrix linear systems.

Arguments num\_threads (int\*) desired number of threads (OpenMP or Pthreads, depending on how SUPERLUMT was installed) to use during the factorization steps.

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

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

The SUNLinSol\_SuperLUMTSetOrdering routine also supports Fortran interfaces for the system and mass matrix solvers:

#### FSUNSUPERLUMTSETORDERING

Call FSUNSUPERLUMTSETORDERING(code, ordering, ier)

Description The function FSUNSUPERLUMTSETORDERING can be called for Fortran programs to update

the ordering algorithm in a SUNLINSOL\_SUPERLUMT object.

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

ordering (int\*) a flag indicating the ordering algorithm, options are:

0 natural ordering

1 minimal degree ordering on  $A^TA$ 

2 minimal degree ordering on  $A^T + A$ 

3 COLAMD ordering for unsymmetric matrices

The default is 3 for COLAMD.

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

Notes See SUNLinSol\_SuperLUMTSetOrdering for complete further documentation of this routine.

#### FSUNMASSUPERLUMTSETORDERING

Call FSUNMASSUPERLUMTSETORDERING(ordering, ier)

The function FSUNMASSUPERLUMTSETORDERING can be called for Fortran programs to Description update the ordering algorithm in a SUNLINSOL\_SUPERLUMT object for mass matrix linear systems.

Arguments ordering (int\*) a flag indicating the ordering algorithm, options are:

0 natural ordering

```
1 minimal degree ordering on A^TA
2 minimal degree ordering on A^T + A
3 COLAMD ordering for unsymmetric matrices
```

The default is 3 for COLAMD.

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

Notes See SUNLinSol\_SuperLUMTSetOrdering for complete further documentation of this routine.

## 9.11.4 SUNLinearSolver\_SuperLUMT content

The SUNLINSOL\_SUPERLUMT module defines the *content* field of a SUNLinearSolver as the following structure:

```
struct _SUNLinearSolverContent_SuperLUMT {
  int
                last_flag;
  int
                first_factorize;
  SuperMatrix *A, *AC, *L, *U, *B;
  Gstat_t
                *Gstat;
  sunindextype *perm_r, *perm_c;
  sunindextype N;
  int
                num_threads;
  realtype
                diag_pivot_thresh;
  int
                ordering;
  superlumt_options_t *options;
};
These entries of the content field contain the following information:
last_flag
                   - last error return flag from internal function evaluations,
first_factorize
                  - flag indicating whether the factorization has ever been performed,
A, AC, L, U, B
                   - SuperMatrix pointers used in solve,
Gstat
                   - GStat_t object used in solve,
perm_r, perm_c
                   - permutation arrays used in solve,
                   - size of the linear system,
num_threads
                   - number of OpenMP/Pthreads threads to use,
diag_pivot_thresh - threshold on diagonal pivoting,
                   - flag for which reordering algorithm to use,
ordering
```

- pointer to SUPERLUMT options structure.

options

# 9.12 The SUNLinearSolver\_cuSolverSp\_batchQR implementation

The SUNLinearSolver\_cuSolverSp\_batchQR implementation of the SUNLINSOL API is designed to be used with the SUNMATRIX\_CUSPARSE matrix, and the NVECTOR\_CUDA vector. The header file to include when using this module is sunlinsol\_cusolversp\_batchqr.h. The installed library to link to is libsundials\_sunlinsolcusolversp.lib where .lib is typically .so for shared libraries and .a for static libraries.

The SUNLinearSolver\_cuSolverSp\_batchQR module is experimental and subject to change.



## 9.12.1 SUNLinearSolver\_cuSolverSp\_batchQR description

The SUNLinearSolver\_cuSolverSp\_batchQR implementation provides an interface to the batched sparse QR factorization method provided by the NVIDIA cuSOLVER library [6]. The module is designed for solving block diagonal linear systems of the form

$$\begin{bmatrix} \mathbf{A_1} & 0 & \cdots & 0 \\ 0 & \mathbf{A_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A_n} \end{bmatrix} x_j = b_j$$

where all block matrices  $\mathbf{A_j}$  share the same sparsisty pattern. The matrix must be the SUNMATRIX\_CUSPARSE module.

## 9.12.2 SUNLinearSolver\_cuSolverSp\_batchQR functions

The SUNLinearSolver\_cuSolverSp\_batchQR module defines implementations of all "direct" linear solver operations listed in Sections 9.1.1-9.1.3:

- SUNLinSolGetType\_cuSolverSp\_batchQR
- SUNLinSolInitialize\_cuSolverSp\_batchQR this sets the first\_factorize flag to 1
- SUNLinSolSetup\_cuSolverSp\_batchQR this always copies the relevant SUNMATRIX\_SPARSE data to the GPU; if this is the first setup it will perform symbolic analysis on the system
- SUNLinSolSolve\_cuSolverSp\_batchQR this calls the cusolverSpXcsrqrsvBatched routine to perform factorization
- SUNLinSolLastFlag\_cuSolverSp\_batchQR
- SUNLinSolFree\_cuSolverSp\_batchQR

In addition, the module provides the following user-callable routines:

#### SUNLinSol\_cuSolverSp\_batchQR

Call LS = SUNLinSol\_cuSolverSp\_batchQR(y, A, cusol);

Description The function SUNLinSol\_cuSolverSp\_batchQR creates and allocates memory for a SUNLINSOL object.

Arguments y (N\_Vector) a NVECTOR\_CUDA vector for checking compatibility with the

A (SUNMatrix) a SUNMATRIX\_SPARSE matrix for checking compatibility with the solver

ol (cusolverHandle\_t) a valid cuSOLVER handle

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

Notes This routine analyzes the input matrix and vector to determine the linear system size and to assess compatibility with the solver.

This routine will perform consistency checks to ensure that it is called with consistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMAT\_CUSPARSE matrix type and the NVECTOR\_CUDA vector type. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

```
|SUNLinSol_cuSolverSp_batchQR_GetDescription
Call
             SUNLinSol_cuSolverSp_batchQR_GetDescription(LS, &desc);
Description
             The function SUNLinSol_cuSolverSp_batchQR_GetDescription accesses the string de-
             scription of the object (empty by default).
Arguments
             LS
                      (SUNLinearSolver) a SUNLinSol_cuSolverSp_batchQR object
             desc
                      (char **) the string description of the linear solver
Return value None
SUNLinSol_cuSolverSp_batchQR_SetDescription
Call
             SUNLinSol_cuSolverSp_batchQR_SetDescription(LS, desc);
             The function SUNLinSol_cuSolverSp_batchQR_SetDescription sets the string descrip-
Description
             tion of the object (empty by default).
             LS
                      (SUNLinearSolver) a SUNLinSol_cuSolverSp_batchQR object
Arguments
             desc
                      (const char *) the string description of the linear solver
Return value None
SUNLinSol_cuSolverSp_batchQR_GetDeviceSpace
Call
             SUNLinSol_cuSolverSp_batchQR_GetDeviceSpace(LS, cuSolverInternal, cuSolverWorkspace);
Description
             The function \verb|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.
Arguments
                      (SUNLinearSolver) a SUNLinSol_cuSolverSp_batchQR object
             LS
             cuSolverInternal (size_t *) output - the size of the cuSOLVER internal buffer in
                      bytes
             cuSolverWorkspace (size_t *) output - the size of the cuSOLVER workspace buffer
                      in bytes
```

Return value None

#### 9.12.3 SUNLinearSolver\_cuSolverSp\_batchQR content

The SUNLinearSolver\_cuSolverSp\_batchQR module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_cuSolverSp_batchQR {
  int
                     last_flag;
                                         /* last return flag
                                                                                                  */
                                         /* is this the first factorization?
  booleantype
                     first_factorize;
                                                                                                  */
                                                                                                  */
                                         /* size of cusolver internal buffer for Q and R
  size_t
                     internal_size;
                     workspace_size;
                                         /* size of cusolver memory block for num. factorization
                                                                                                  */
  size t
  cusolverSpHandle_t cusolver_handle;
                                         /* cuSolverSp context
                                                                                                  */
  csrqrInfo_t
                     info;
                                         /* opaque cusolver data structure
                                                                                                  */
                                         /* memory block used by cusolver
  void*
                     workspace;
                                                                                                  */
                     desc;
  const char*
                                         /* description of this linear solver
};
```

## 9.13 The SUNLinearSolver\_MagmaDense implementation

The SUNLinearSolver\_MagmaDense implementation of the SUNLINSOL API is designed to be used with the SUNMATRIX\_MAGMADENSE matrix, and a GPU-enabled vector. This implementation interfaces to the MAGMA () linear algebra library and can target NVIDIA's CUDA programming model or AMD's HIP programming model [45].

The header file to include when using this module is sunlinsol/sunlinsol\_magmadense.h. The installed library to link to is libsundials\_sunlinsolmagmadense.lib where .lib is typically .so for shared libraries and .a for static libraries.



The SUNLinearSolver\_MagmaDense module is experimental and subject to change.

#### 9.13.1 SUNLinearSolver\_MagmaDense description

The SUNLinearSolver\_MagmaDense implementation provides an interface to the dense LU and dense batched LU methods in the MAGMA linear algebra library [4]. The batched LU methods are leveraged when solving block diagonal linear systems of the form

$$\begin{bmatrix} \mathbf{A_0} & 0 & \cdots & 0 \\ 0 & \mathbf{A_1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A_{n-1}} \end{bmatrix} x_j = b_j.$$

## 9.13.2 SUNLinearSolver\_MagmaDense functions

The SUNLinearSolver\_MagmaDense module defines implementations of all "direct" linear solver operations listed in Sections 9.1.1-9.1.3:

- SUNLinSolGetType\_MagmaDense
- SUNLinSolInitialize\_MagmaDense
- SUNLinSolSetup\_MagmaDense
- SUNLinSolSolve\_MagmaDense
- SUNLinSolLastFlag\_MagmaDense
- SUNLinSolFree\_MagmaDense

In addition, the module provides the following user-callable routines:

${\tt SUNLinSol\_MagmaDense}$

Call LS = SUNLinSol\_MagmaDense(y, A);

Description The function SUNLinSol\_MagmaDense creates and allocates memory for a SUNLINSOL object.

Arguments y (N\_Vector) a vector for checking compatibility with the solver

A (SUNMatrix) a SUNMATRIX\_MAGMADENSE matrix for checking compatibility with the solver

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

Notes This routine analyzes the input matrix and vector to determine the linear system size and to assess compatibility with the solver.

```
Call SUNLinSol_MagmaDense_SetAsync(SUNLinearSolver LS, booleantype onoff);

Description The function SUNLinSol_MagmaDense_SetAsync can be used to toggle the linear solver between asynchronous and synchronous modes. In asynchronous mode, SUNLinearSolver operations are asynchronous with respect to the host. In synchronous mode, the host and GPU device are synchronized prior to the operation returning.

Arguments LS (SUNLinearSolver) a SUNLinSol_MagmaDense object onoff (booleantype) set to 0 for synchronous mode, or 1 for asynchronous mode

Return value None

The default is asynchronous mode.
```

#### 9.13.3 SUNLinearSolver\_MagmaDense content

The SUNLinearSolver\_MagmaDense module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_MagmaDense {
  int
                  last_flag;
  booleantype
                  async;
  sunindextype
                  N;
  SUNMemory
                  pivots;
  SUNMemory
                  pivotsarr;
  SUNMemory
                  dpivotsarr;
  SUNMemory
                  infoarr;
  SUNMemory
                  rhsarr;
  SUNMemoryHelper memhelp;
  magma_queue_t
};
```

## 9.14 The SUNLinearSolver\_SPGMR implementation

This section describes the SUNLINSOL implementation of the SPGMR (Scaled, Preconditioned, Generalized Minimum Residual [44]) iterative linear solver. The SUNLINSOL\_SPGMR module is designed to be compatible with any NVECTOR implementation that supports a minimal subset of operations (N\_VClone, N\_VDotProd, N\_VScale, N\_VLinearSum, N\_VProd, N\_VConst, N\_VDiv, and N\_VDestroy). When using Classical Gram-Schmidt, the optional function N\_VDotProdMulti may be supplied for increased efficiency.

To access the SUNLINSOL\_SPGMR module, include the header file sunlinsol/sunlinsol\_spgmr.h. We note that the SUNLINSOL\_SPGMR module is accessible from SUNDIALS packages without separately linking to the libsundials\_sunlinsolspgmr module library.

#### 9.14.1 SUNLinearSolver\_SPGMR description

This solver is constructed to perform the following operations:

- During construction, the xcor and vtemp arrays are cloned from a template NVECTOR that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLINSOL\_SPGMR to supply the ATimes, PSetup, and Psolve function pointers and s1 and s2 scaling vectors.
- In the "initialize" call, the remaining solver data is allocated (V, Hes, givens, and yg)

- In the "setup" call, any non-NULL PSetup function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic PSetup function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call, the GMRES iteration is performed. This will include scaling, preconditioning, and restarts if those options have been supplied.

#### 9.14.2 SUNLinearSolver\_SPGMR functions

The SUNLINSOL\_SPGMR module provides the following user-callable constructor for creating a SUNLinearSolver object.

#### SUNLinSol\_SPGMR

Notes

Call LS = SUNLinSol\_SPGMR(y, pretype, maxl);

Description The function SUNLinSol\_SPGMR creates and allocates memory for a SPGMR

SUNLinearSolver object.

Arguments y (N\_Vector) a template for cloning vectors needed within the solver

pretype (int) flag indicating the desired type of preconditioning, allowed values

- PREC\_NONE (0)
- PREC\_LEFT (1)
- PREC\_RIGHT (2)
- PREC\_BOTH (3)

Any other integer input will result in the default (no preconditioning).

(int) the number of Krylov basis vectors to use. Values  $\leq 0$  will result in the default value (5).

Return value This returns a SUNLinearSolver object. If either y is incompatible then this

routine will return NULL.

This routine will perform consistency checks to ensure that it is called with a consistent NVECTOR implementation (i.e. that it supplies the requisite vector operations). If y is incompatible, then this routine will return NULL.

We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL). While it is possible to configure a SUNLINSOL\_SPGMR object to use any of the preconditioning options with these solvers, this use mode is not supported and may result

in inferior performance.

Deprecated Name For backward compatibility, the wrapper function SUNSPGMR with idential input and output arguments is also provided.

F2003 Name FSUNLinSol\_SPGMR

maxl

The SUNLINSOL\_SPGMR module defines implementations of all "iterative" linear solver operations listed in Sections 9.1.1 - 9.1.3:

- SUNLinSolGetType\_SPGMR
- SUNLinSolInitialize\_SPGMR
- SUNLinSolSetATimes\_SPGMR
- SUNLinSolSetPreconditioner\_SPGMR
- SUNLinSolSetScalingVectors\_SPGMR

- SUNLinSolSetup\_SPGMR
- SUNLinSolSolve\_SPGMR
- SUNLinSolNumIters\_SPGMR
- SUNLinSolResNorm\_SPGMR
- SUNLinSolResid\_SPGMR
- SUNLinSolLastFlag\_SPGMR
- SUNLinSolSpace\_SPGMR
- SUNLinSolFree\_SPGMR

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an 'F' to the function name.

The SUNLINSOL\_SPGMR module also defines the following additional user-callable functions.

#### SUNLinSol\_SPGMRSetPrecType

Call retval = SUNLinSol\_SPGMRSetPrecType(LS, pretype);

Description The function SUNLinSol\_SPGMRSetPrecType updates the type of preconditioning

to use in the SUNLINSOL\_SPGMR object.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SPGMR object to update

pretype (int) flag indicating the desired type of preconditioning, allowed values

 $match\ those\ discussed\ in\ {\tt SUNLinSol\_SPGMR}.$ 

Return value This routine will return with one of the error codes SUNLS\_ILL\_INPUT (illegal

pretype), SUNLS\_MEM\_NULL (S is NULL) or SUNLS\_SUCCESS.

Deprecated Name For backward compatibility, the wrapper function SUNSPGMRSetPrecType with

idential input and output arguments is also provided.

F2003 Name FSUNLinSol\_SPGMRSetPrecType

#### |SUNLinSol\_SPGMRSetGSType

Call retval = SUNLinSol\_SPGMRSetGSType(LS, gstype);

Description The function SUNLinSol\_SPGMRSetPrecType sets the type of Gram-Schmidt or-

thogonalization to use in the SUNLINSOL\_SPGMR object.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SPGMR object to update

gstype (int) flag indicating the desired orthogonalization algorithm; allowed val-

ues are:

• MODIFIED\_GS (1)

• CLASSICAL\_GS (2)

Any other integer input will result in a failure, returning error code SUNLS\_ILL\_INPUT.

Return value This routine will return with one of the error codes SUNLS\_ILL\_INPUT (illegal pretype), SUNLS\_MEM\_NULL (S is NULL) or SUNLS\_SUCCESS.

Deprecated Name For backward compatibility, the wrapper function SUNSPGMRSetGSType with iden-

tial input and output arguments is also provided.

F2003 Name FSUNLinSol\_SPGMRSetGSType

#### SUNLinSol\_SPGMRSetMaxRestarts

Call retval = SUNLinSol\_SPGMRSetMaxRestarts(LS, maxrs);

Description The function SUNLinSol\_SPGMRSetMaxRestarts sets the number of GMRES restarts

to allow in the SUNLINSOL\_SPGMR object.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SPGMR object to update

maxrs (int) integer indicating number of restarts to allow. A negative input will

result in the default of 0.

Return value This routine will return with one of the error codes SUNLS\_MEM\_NULL (S is NULL) or

SUNLS SUCCESS.

Deprecated Name For backward compatibility, the wrapper function SUNSPGMRSetMaxRestarts with

idential input and output arguments is also provided.

F2003 Name FSUNLinSol\_SPGMRSetMaxRestarts

#### SUNLinSolSetInfoFile\_SPGMR

Call retval = SUNLinSolSetInfoFile\_SPGMR(LS, info\_file);

Description The function SUNLinSolSetInfoFile\_SPGMR sets the output file where all informative

(non-error) messages should be directed.

Arguments LS (SUNLinearSolver) a SUNNONLINSOL object

info\_file (FILE\*) pointer to output file (stdout by default); a NULL input will disable

outpu

Return value The return value is

• SUNLS\_SUCCESS if successful

• SUNLS\_MEM\_NULL if the SUNLinearSolver memory was NULL

• SUNLS\_ILL\_INPUT if SUNDIALS was not built with monitoring enabled

Notes This function is intended for users that wish to monitor the linear solver progress. By

default, the file pointer is set to stdout.

SUNDIALS must be built with the CMake option SUNDIALS\_BUILD\_WITH\_MONITORING, to utilize this function. See section A.1.2 for more information.

F2003 Name FSUNLinSolSetInfoFile\_SPGMR

#### SUNLinSolSetPrintLevel\_SPGMR

Call retval = SUNLinSolSetPrintLevel\_SPGMR(NLS, print\_level);

Description The function SUNLinSolSetPrintLevel\_SPGMR specifies the level of verbosity of the

output.

Arguments LS (SUNLinearSolver) a SUNNONLINSOL object

print\_level (int) flag indicating level of verbosity; must be one of:

- 0, no information is printed (default)
- 1, for each linear iteration the residual norm is printed

Return value The return value is

- SUNLS\_SUCCESS if successful
- SUNLS\_MEM\_NULL if the SUNLinearSolver memory was NULL
- SUNLS\_ILL\_INPUT if SUNDIALS was not built with monitoring enabled, or the print level value was invalid

Notes This function is intended for users that wish to monitor the linear solver progress. By

default, the print level is 0.

SUNDIALS must be built with the CMake option SUNDIALS\_BUILD\_WITH\_MONITORING,

to utilize this function. See section A.1.2 for more information.

F2003 Name FSUNLinSolSetPrintLevel\_SPGMR

#### 9.14.3 SUNLinearSolver\_SPGMR Fortran interfaces

The SUNLINSOL\_SPGMR module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

#### FORTRAN 2003 interface module

The fsunlinsol\_spgmr\_mod FORTRAN module defines interfaces to all SUNLINSOL\_SPGMR C functions using the intrinsic iso\_c\_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function SUNLinSol\_SPGMR is interfaced as FSUNLinSol\_SPGMR.

The Fortran 2003 sunlinsol\_spgmr interface module can be accessed with the use statement, i.e. use fsunlinsol\_spgmr\_mod, and linking to the library libsundials\_fsunlinsolspgmr\_mod.lib in addition to the C library. For details on where the library and module file fsunlinsol\_spgmr\_mod.mod are installed see Appendix A. We note that the module is accessible from the Fortran 2003 sundials integrators without separately linking to the libsundials\_fsunlinsolspgmr\_mod library.

#### FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL\_SPGMR module also includes a Fortran-callable function for creating a SUNLinearSolver object.

#### FSUNSPGMRINIT

Call FSUNSPGMRINIT(code, pretype, maxl, ier)

Description The function FSUNSPGMRINIT can be called for Fortran programs to create a SUNLIN-

SOL\_SPGMR object.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3

for KINSOL, and 4 for ARKODE).

pretype (int\*) flag indicating desired preconditioning type

maxl (int\*) flag indicating Krylov subspace size

Return value ier is a return completion flag equal to 0 for a success return and -1 otherwise. See

printed message for details in case of failure.

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

Allowable values for pretype and max1 are the same as for the C function

SUNLinSol\_SPGMR.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL\_SPGMR module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

#### FSUNMASSSPGMRINIT

Call FSUNMASSSPGMRINIT(pretype, maxl, ier)

Description The function FSUNMASSSPGMRINIT can be called for Fortran programs to create a SUN-

LINSOL\_SPGMR object for mass matrix linear systems.

Arguments pretype (int\*) flag indicating desired preconditioning type

maxl (int\*) flag indicating Krylov subspace size

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

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

Allowable values for pretype and max1 are the same as for the C function

SUNLinSol\_SPGMR.

The SUNLinSol\_SPGMRSetPrecType, SUNLinSol\_SPGMRSetGSType and

SUNLinSol\_SPGMRSetMaxRestarts routines also support Fortran interfaces for the system and mass matrix solvers.

#### FSUNSPGMRSETGSTYPE

Call FSUNSPGMRSETGSTYPE(code, gstype, ier)

Description The function FSUNSPGMRSETGSTYPE can be called for Fortran programs to change the

Gram-Schmidt orthogonaliation algorithm.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for

KINSOL, and 4 for ARKODE).

gstype (int\*) flag indicating the desired orthogonalization algorithm.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPGMRSetGSType for complete further documentation of this routine.

#### FSUNMASSSPGMRSETGSTYPE

Call FSUNMASSSPGMRSETGSTYPE(gstype, ier)

Description The function FSUNMASSSPGMRSETGSTYPE can be called for Fortran programs to change

the Gram-Schmidt orthogonaliation algorithm for mass matrix linear systems.

Arguments The arguments are identical to FSUNSPGMRSETGSTYPE above, except that code is not

needed since mass matrix linear systems only arise in ARKODE.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPGMRSetGSType for complete further documentation of this routine.

#### FSUNSPGMRSETPRECTYPE

Call FSUNSPGMRSETPRECTYPE(code, pretype, ier)

Description The function FSUNSPGMRSETPRECTYPE can be called for Fortran programs to change the

type of preconditioning to use.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3

for KINSOL, and 4 for ARKODE).

pretype (int\*) flag indicating the type of preconditioning to use.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPGMRSetPrecType for complete further documentation of this routine.

#### FSUNMASSSPGMRSETPRECTYPE

Call FSUNMASSSPGMRSETPRECTYPE(pretype, ier)

Description The function FSUNMASSSPGMRSETPRECTYPE can be called for Fortran programs to change

the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to FSUNSPGMRSETPRECTYPE above, except that code is not

needed since mass matrix linear systems only arise in ARKODE.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPGMRSetPrecType for complete further documentation of this routine.

#### FSUNSPGMRSETMAXRS

Call FSUNSPGMRSETMAXRS(code, maxrs, ier)

Description The function FSUNSPGMRSETMAXRS can be called for Fortran programs to change the

maximum number of restarts allowed for SPGMR.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for

KINSOL, and 4 for ARKODE).

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

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPGMRSetMaxRestarts for complete further documentation of this rou-

tine.

#### FSUNMASSSPGMRSETMAXRS

Call FSUNMASSSPGMRSETMAXRS(maxrs, ier)

Description The function FSUNMASSSPGMRSETMAXRS can be called for Fortran programs to change

the maximum number of restarts allowed for SPGMR for mass matrix linear systems.

Arguments The arguments are identical to FSUNSPGMRSETMAXRS above, except that code is not

needed since mass matrix linear systems only arise in ARKODE.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPGMRSetMaxRestarts for complete further documentation of this rou-

tine

#### 9.14.4 SUNLinearSolver\_SPGMR content

The SUNLINSOL\_SPGMR module defines the *content* field of a SUNLinearSolver as the following structure:

```
struct _SUNLinearSolverContent_SPGMR {
  int maxl;
  int pretype;
  int gstype;
  int max_restarts;
  int numiters;
  realtype resnorm;
  int last_flag;
  ATimesFn ATimes;
  void* ATData;
  PSetupFn Psetup;
```

```
PSolveFn Psolve;
  void* PData;
  N_Vector s1;
  N_Vector s2;
  N_Vector *V;
  realtype **Hes;
  realtype *givens;
  N_Vector xcor;
  realtype *yg;
  N_Vector vtemp;
  int
             print_level;
  FILE*
             info_file;
};
These entries of the content field contain the following information:
               - number of GMRES basis vectors to use (default is 5),
pretype
               - flag for type of preconditioning to employ (default is none),
               - flag for type of Gram-Schmidt orthogonalization (default is modified Gram-Schmidt),
gstype
max_restarts - number of GMRES restarts to allow (default is 0),
numiters
               - number of iterations from the most-recent solve,
               - final linear residual norm from the most-recent solve,
resnorm
               - last error return flag from an internal function,
last_flag
ATimes
               - function pointer to perform Av product,
               - pointer to structure for ATimes,
ATData
               - function pointer to preconditioner setup routine,
Psetup
Psolve
               - function pointer to preconditioner solve routine,
PData
               - pointer to structure for Psetup and Psolve,
               - vector pointers for supplied scaling matrices (default is NULL),
s1, s2
               - the array of Krylov basis vectors v_1,\ldots,v_{\mathtt{maxl}+1}, stored in \mathtt{V[0]},\ldots,\mathtt{V[maxl]}. Each
               v_i is a vector of type NVECTOR.,
               - the (maxl + 1) × maxl Hessenberg matrix. It is stored row-wise so that the (i,j)th
Hes
               element is given by Hes[i][j].,
               - a length 2*maxl array which represents the Givens rotation matrices that arise in the
givens
               GMRES algorithm. These matrices are F_0, F_1, \ldots, F_j, where
```

```
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,

- a length (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 fileinfo\_file - the file where all informative (non-error) messages will be directed

## 9.15 The SUNLinearSolver\_SPFGMR implementation

This section describes the SUNLINSOL implementation of the SPFGMR (Scaled, Preconditioned, Flexible, Generalized Minimum Residual [43]) iterative linear solver. The SUNLINSOL\_SPFGMR module is designed to be compatible with any NVECTOR implementation that supports a minimal subset of operations (N\_VClone, N\_VDotProd, N\_VScale, N\_VLinearSum, N\_VProd, N\_VConst, N\_VDiv, and N\_VDestroy). When using Classical Gram-Schmidt, the optional function N\_VDotProdMulti may be supplied for increased efficiency. Unlike the other Krylov iterative linear solvers supplied with SUNDIALS, SPFGMR is specifically designed to work with a changing preconditioner (e.g. from an iterative method).

To access the SUNLINSOL\_SPFGMR module, include the header file sunlinsol/sunlinsol\_spfgmr.h. We note that the SUNLINSOL\_SPFGMR module is accessible from SUNDIALS packages without separately linking to the libsundials\_sunlinsolspfgmr module library.

## 9.15.1 SUNLinearSolver\_SPFGMR description

This solver is constructed to perform the following operations:

- During construction, the xcor and vtemp arrays are cloned from a template NVECTOR that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLINSOL\_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.

#### 9.15.2 SUNLinearSolver\_SPFGMR functions

The SUNLINSOL\_SPFGMR module provides the following user-callable constructor for creating a SUNLinearSolver object.

#### SUNLinSol\_SPFGMR

```
Call LS = SUNLinSol_SPFGMR(y, pretype, maxl);
```

Description The function SUNLinSol\_SPFGMR creates and allocates memory for a SPFGMR SUNLinearSolver object.

Arguments y (N\_Vector) a template for cloning vectors needed within the solver pretype (int) flag indicating the desired type of preconditioning, allowed values are:

- PREC\_NONE (0)
- PREC\_LEFT (1)
- PREC\_RIGHT (2)

• PREC\_BOTH (3)

Any other integer input will result in the default (no preconditioning).

maxl (int) the number of Krylov basis vectors to use. Values  $\leq 0$  will result in the default value (5).

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

Notes

This routine will perform consistency checks to ensure that it is called with a consistent NVECTOR implementation (i.e. that it supplies the requisite vector operations). If y is incompatible, then this routine will return NULL.

We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL). While it is possible to configure a SUNLINSOL\_SPFGMR object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.

F2003 Name FSUNLinSol\_SPFGMR

SUNSPFGMR The SUNLINSOL\_SPFGMR module defines implementations of all "iterative" linear solver operations listed in Sections 9.1.1 - 9.1.3:

- SUNLinSolGetType\_SPFGMR
- SUNLinSolInitialize\_SPFGMR
- SUNLinSolSetATimes\_SPFGMR
- SUNLinSolSetPreconditioner\_SPFGMR
- SUNLinSolSetScalingVectors\_SPFGMR
- SUNLinSolSetup\_SPFGMR
- SUNLinSolSolve\_SPFGMR
- SUNLinSolNumIters\_SPFGMR
- SUNLinSolResNorm\_SPFGMR
- SUNLinSolResid\_SPFGMR
- SUNLinSolLastFlag\_SPFGMR
- SUNLinSolSpace\_SPFGMR
- SUNLinSolFree\_SPFGMR

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an 'F' to the function name.

The SUNLINSOL\_SPFGMR module also defines the following additional user-callable functions.

#### | SUNLinSol\_SPFGMRSetPrecType

Call retval = SUNLinSol\_SPFGMRSetPrecType(LS, pretype);

Description The function SUNLinSol\_SPFGMRSetPrecType updates the type of preconditioning

to use in the SUNLINSOL\_SPFGMR object.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SPFGMR object to update

pretype (int) flag indicating the desired type of preconditioning, allowed values

match those discussed in SUNLinSol\_SPFGMR.

Return value This routine will return with one of the error codes SUNLS\_ILL\_INPUT (illegal

pretype), SUNLS\_MEM\_NULL (S is NULL) or SUNLS\_SUCCESS.

Deprecated Name For backward compatibility, the wrapper function SUNSPFGMRSetPrecType with

idential input and output arguments is also provided.

F2003 Name FSUNLinSol\_SPFGMRSetPrecType

SUNLinSol\_SPFGMRSetGSType

Call retval = SUNLinSol\_SPFGMRSetGSType(LS, gstype);

Description The function SUNLinSol\_SPFGMRSetPrecType sets the type of Gram-Schmidt or-

thogonalization to use in the SUNLINSOL\_SPEGMR object.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SPFGMR object to update

gstype (int) flag indicating the desired orthogonalization algorithm; allowed val-

ues are:

• MODIFIED\_GS (1)

• CLASSICAL\_GS (2)

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

SUNLS\_ILL\_INPUT.

Return value This routine will return with one of the error codes SUNLS\_ILL\_INPUT (illegal

pretype), SUNLS\_MEM\_NULL (S is NULL) or SUNLS\_SUCCESS.

Deprecated Name For backward compatibility, the wrapper function SUNSPFGMRSetGSType with iden-

tial input and output arguments is also provided.

F2003 Name FSUNLinSol\_SPFGMRSetGSType

SUNLinSol\_SPFGMRSetMaxRestarts

Call retval = SUNLinSol\_SPFGMRSetMaxRestarts(LS, maxrs);

Description The function SUNLinSol\_SPFGMRSetMaxRestarts sets the number of GMRES

restarts to allow in the SUNLINSOL\_SPFGMR object.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SPFGMR object to update

maxrs (int) integer indicating number of restarts to allow. A negative input will

result in the default of 0.

Return value This routine will return with one of the error codes SUNLS\_MEM\_NULL (S is NULL) or

SUNLS\_SUCCESS.

Deprecated Name For backward compatibility, the wrapper function SUNSPFGMRSetMaxRestarts with

idential input and output arguments is also provided.

F2003 Name FSUNLinSol\_SPFGMRSetMaxRestarts

SUNLinSolSetInfoFile\_SPFGMR

Call retval = SUNLinSolSetInfoFile\_SPFGMR(LS, info\_file);

Description The function SUNLinSolSetInfoFile\_SPFGMR sets the output file where all informative

(non-error) messages should be directed.

Arguments LS (SUNLinearSolver) a SUNNONLINSOL object

 $\verb|info_file| (\verb|FILE*|) pointer to output file (\verb|stdout| by default); a \verb|NULL| input will disable| \\$ 

output

Return value The return value is

• SUNLS\_SUCCESS if successful

• SUNLS\_MEM\_NULL if the SUNLinearSolver memory was NULL

• SUNLS\_ILL\_INPUT if SUNDIALS was not built with monitoring enabled

Notes This function is intended for users that wish to monitor the linear solver progress. By

default, the file pointer is set to stdout.

SUNDIALS must be built with the CMake option SUNDIALS\_BUILD\_WITH\_MONITORING,

to utilize this function. See section A.1.2 for more information.

F2003 Name FSUNLinSolSetInfoFile\_SPFGMR

#### SUNLinSolSetPrintLevel\_SPFGMR

Call retval = SUNLinSolSetPrintLevel\_SPFGMR(NLS, print\_level);

Description The function SUNLinSolSetPrintLevel\_SPFGMR specifies the level of verbosity of the

output.

Arguments LS (SUNLinearSolver) a SUNNONLINSOL object

print\_level (int) flag indicating level of verbosity; must be one of:

• 0, no information is printed (default)

• 1, for each linear iteration the residual norm is printed

Return value The return value is

• SUNLS\_SUCCESS if successful

- SUNLS\_MEM\_NULL if the SUNLinearSolver memory was NULL
- SUNLS\_ILL\_INPUT if SUNDIALS was not built with monitoring enabled, or the print level value was invalid

Notes This function is intended for users that wish to monitor the linear solver progress. By default, the print level is 0.

SUNDIALS must be built with the CMake option SUNDIALS\_BUILD\_WITH\_MONITORING, to utilize this function. See section A.1.2 for more information.

F2003 Name FSUNLinSolSetPrintLevel\_SPFGMR

#### 9.15.3 SUNLinearSolver SPFGMR Fortran interfaces

The SUNLINSOL\_SPFGMR module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

#### FORTRAN 2003 interface module

The fsunlinsol\_spfgmr\_mod FORTRAN module defines interfaces to all SUNLINSOL\_SPFGMR C functions using the intrinsic iso\_c\_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function SUNLinSol\_SPFGMR is interfaced as FSUNLinSol\_SPFGMR.

The Fortran 2003 sunlinsol\_spfgmr interface module can be accessed with the use statement, i.e. use fsunlinsol\_spfgmr\_mod, and linking to the library libsundials\_fsunlinsolspfgmr\_mod. lib in addition to the C library. For details on where the library and module file

fsunlinsol\_spfgmr\_mod.mod are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the libsundials\_fsunlinsolspfgmr\_mod library.

#### FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL\_SPFGMR module also includes a Fortran-callable function for creating a SUNLinearSolver object.

#### FSUNSPFGMRINIT

Call FSUNSPFGMRINIT(code, pretype, maxl, ier)

Description The function FSUNSPFGMRINIT can be called for Fortran programs to create a SUNLIN-

SOL\_SPFGMR object.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3

for KINSOL, and 4 for ARKODE).

pretype (int\*) flag indicating desired preconditioning type

maxl (int\*) flag indicating Krylov subspace size

Return value ier is a return completion flag equal to 0 for a success return and -1 otherwise. See

printed message for details in case of failure.

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

Allowable values for pretype and max1 are the same as for the C function

SUNLinSol\_SPFGMR.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL\_SPFGMR module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

#### FSUNMASSSPFGMRINIT

Call FSUNMASSSPFGMRINIT(pretype, maxl, ier)

Description The function FSUNMASSSPFGMRINIT can be called for Fortran programs to create a SUN-

LINSOL\_SPFGMR object for mass matrix linear systems.

Arguments pretype (int\*) flag indicating desired preconditioning type

maxl (int\*) flag indicating Krylov subspace size

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

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

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

SUNLinSol\_SPFGMR.

The SUNLinSol\_SPFGMRSetPrecType, SUNLinSol\_SPFGMRSetGSType and

SUNLinSol\_SPFGMRSetMaxRestarts routines also support Fortran interfaces for the system and mass matrix solvers.

#### FSUNSPFGMRSETGSTYPE

Call FSUNSPFGMRSETGSTYPE(code, gstype, ier)

Description The function FSUNSPFGMRSETGSTYPE can be called for Fortran programs to change the

 ${\bf Gram\text{-}Schmidt\ orthogonaliation\ algorithm.}$ 

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for

KINSOL, and 4 for ARKODE).

gstype (int\*) flag indicating the desired orthogonalization algorithm.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPFGMRSetGSType for complete further documentation of this routine.

#### FSUNMASSSPFGMRSETGSTYPE

Call FSUNMASSSPFGMRSETGSTYPE(gstype, ier)

Description The function FSUNMASSSPFGMRSETGSTYPE can be called for Fortran programs to change

the Gram-Schmidt orthogonaliation algorithm for mass matrix linear systems.

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

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

Notes See SUNLinSol\_SPFGMRSetGSType for complete further documentation of this routine.

#### FSUNSPFGMRSETPRECTYPE

Call FSUNSPFGMRSETPRECTYPE(code, pretype, ier)

Description The function FSUNSPFGMRSETPRECTYPE can be called for Fortran programs to change

the type of preconditioning to use.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3

for KINSOL, and 4 for ARKODE).

pretype (int\*) flag indicating the type of preconditioning to use.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPFGMRSetPrecType for complete further documentation of this routine.

#### FSUNMASSSPFGMRSETPRECTYPE

Call FSUNMASSSPFGMRSETPRECTYPE(pretype, ier)

Description The function FSUNMASSSPFGMRSETPRECTYPE can be called for Fortran programs to change

the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to FSUNSPFGMRSETPRECTYPE above, except that code is not

needed since mass matrix linear systems only arise in ARKODE.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPFGMRSetPrecType for complete further documentation of this routine.

#### FSUNSPFGMRSETMAXRS

Call FSUNSPFGMRSETMAXRS(code, maxrs, ier)

Description The function FSUNSPFGMRSETMAXRS can be called for Fortran programs to change the

maximum number of restarts allowed for SPFGMR.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for

KINSOL, and 4 for ARKODE).

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

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPFGMRSetMaxRestarts for complete further documentation of this rou-

tine.

#### FSUNMASSSPFGMRSETMAXRS

Call FSUNMASSSPFGMRSETMAXRS(maxrs, ier)

Description The function FSUNMASSSPFGMRSETMAXRS can be called for Fortran programs to change

the maximum number of restarts allowed for SPFGMR for mass matrix linear systems.

Arguments The arguments are identical to FSUNSPFGMRSETMAXRS above, except that code is not

needed since mass matrix linear systems only arise in ARKODE.

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

See SUNLinSol\_SPFGMRSetMaxRestarts for complete further documentation of this rou-

tine.

Notes

#### 9.15.4 SUNLinearSolver\_SPFGMR content

The SUNLINSOL\_SPFGMR module defines the *content* field of a SUNLinearSolver as the following structure:

```
struct _SUNLinearSolverContent_SPFGMR {
  int maxl;
  int pretype;
  int gstype;
  int max_restarts;
  int numiters;
  realtype resnorm;
  int last_flag;
  ATimesFn ATimes;
  void* ATData;
  PSetupFn Psetup;
  PSolveFn Psolve;
  void* PData;
  N_Vector s1;
  N_Vector s2;
  N_Vector *V;
  N_Vector *Z;
  realtype **Hes;
  realtype *givens;
  N_Vector xcor;
  realtype *yg;
  N_Vector vtemp;
  int
            print_level;
  FILE*
            info_file;
};
These entries of the content field contain the following information:
              - number of FGMRES basis vectors to use (default is 5),
maxl
              - flag for type of preconditioning to employ (default is none),
pretype
              - flag for type of Gram-Schmidt orthogonalization (default is modified Gram-Schmidt),
gstype
max_restarts - number of FGMRES restarts to allow (default is 0),
numiters
              - number of iterations from the most-recent solve,
              - final linear residual norm from the most-recent solve,
resnorm
              - last error return flag from an internal function,
last_flag
              - function pointer to perform Av product,
ATimes
              - pointer to structure for ATimes,
ATData
              - function pointer to preconditioner setup routine,
Psetup
Psolve
              - function pointer to preconditioner solve routine,
PData
              - pointer to structure for Psetup and Psolve,
              - vector pointers for supplied scaling matrices (default is NULL),
s1, s2
```

givens

V - the array of Krylov basis vectors  $v_1, \ldots, v_{\mathtt{maxl}+1}$ , stored in  $V[0], \ldots, V[\mathtt{maxl}]$ . Each  $v_i$  is a vector of type NVECTOR.,

Z - the array of preconditioned Krylov basis vectors  $z_1, \ldots, z_{\texttt{maxl}+1}$ , stored in Z[0], ..., Z[maxl]. Each  $z_i$  is a vector of type NVECTOR.,

Hes - the  $(\max 1 + 1) \times \max 1$  Hessenberg matrix. It is stored row-wise so that the (i,j)th element is given by Hes[i][j].,

- a length 2\*maxl array which represents the Givens rotation matrices that arise in the FGMRES algorithm. These matrices are  $F_0, F_1, \ldots, F_j$ , where

are represented in the givens vector as givens [0] =  $c_0$ , givens [1] =  $s_0$ , givens [2]

=  $c_1$ , givens[3] =  $s_1$ , ... givens[2j] =  $c_j$ , givens[2j+1] =  $s_j$ .

**xcor** - a vector which holds the scaled, preconditioned correction to the initial guess,

yg - a length (maxl+1) array of realtype values used to hold "short" vectors (e.g. y and

g),

vtemp - temporary vector storage.

print\_level - controls the amount of information to be printed to the info fileinfo\_file - the file where all informative (non-error) messages will be directed

## 9.16 The SUNLinearSolver\_SPBCGS implementation

This section describes the SUNLINSOL implementation of the SPBCGS (Scaled, Preconditioned, Bi-Conjugate Gradient, Stabilized [46]) iterative linear solver. The SUNLINSOL\_SPBCGS module is designed to be compatible with any NVECTOR implementation that supports a minimal subset of operations (N\_VClone, N\_VDotProd, N\_VScale, N\_VLinearSum, N\_VProd, N\_VDiv, and N\_VDestroy). Unlike the SPGMR and SPFGMR algorithms, SPBCGS requires a fixed amount of memory that does not increase with the number of allowed iterations.

To access the SUNLINSOL\_SPBCGS module, include the header file sunlinsol/sunlinsol\_spbcgs.h. We note that the SUNLINSOL\_SPBCGS module is accessible from SUNDIALS packages without separately linking to the libsundials\_sunlinsolspbcgs module library.

#### 9.16.1 SUNLinearSolver\_SPBCGS description

This solver is constructed to perform the following operations:

- During construction all NVECTOR solver data is allocated, with vectors cloned from a template NVECTOR that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLINSOL\_SPBCGS to supply the ATimes, PSetup, and Psolve function pointers and s1 and s2 scaling vectors.
- In the "initialize" call, the solver parameters are checked for validity.

- In the "setup" call, any non-NULL PSetup function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic PSetup function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call the SPBCGS iteration is performed. This will include scaling and preconditioning if those options have been supplied.

#### 9.16.2 SUNLinearSolver\_SPBCGS functions

The SUNLINSOL\_SPBCGS module provides the following user-callable constructor for creating a SUNLinearSolver object.

#### SUNLinSol\_SPBCGS

Notes

Call LS = SUNLinSol\_SPBCGS(y, pretype, maxl);

Description The function SUNLinSol\_SPBCGS creates and allocates memory for a SPBCGS

SUNLinearSolver object.

Arguments  $y = (N_{\text{-}}Vector)$  a template for cloning vectors needed within the solver pretype (int) flag indicating the desired type of preconditioning, allowed values

• PREC\_NONE (0)

• PREC\_LEFT (1)

• PREC\_RIGHT (2)

• PREC\_BOTH (3)

Any other integer input will result in the default (no preconditioning). (int) the number of linear iterations to allow. Values  $\leq 0$  will result in

the default value (5).

Return value This returns a SUNLinearSolver object. If either y is incompatible then this

routine will return NULL.

This routine will perform consistency checks to ensure that it is called with a consistent NVECTOR implementation (i.e. that it supplies the requisite vector operations). If y is incompatible, then this routine will return NULL.

We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL). While it is possible to configure a SUNLINSOL\_SPBCGS object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.

Deprecated Name For backward compatibility, the wrapper function SUNSPBCGS with idential input and output arguments is also provided.

F2003 Name FSUNLinSol\_SPBCGS

maxl

The SUNLINSOL\_SPBCGS module defines implementations of all "iterative" linear solver operations listed in Sections 9.1.1 - 9.1.3:

- SUNLinSolGetType\_SPBCGS
- SUNLinSolInitialize\_SPBCGS
- SUNLinSolSetATimes\_SPBCGS
- SUNLinSolSetPreconditioner\_SPBCGS
- SUNLinSolSetScalingVectors\_SPBCGS

- SUNLinSolSetup\_SPBCGS
- SUNLinSolSolve\_SPBCGS
- SUNLinSolNumIters\_SPBCGS
- SUNLinSolResNorm\_SPBCGS
- SUNLinSolResid SPBCGS
- SUNLinSolLastFlag\_SPBCGS
- SUNLinSolSpace\_SPBCGS
- SUNLinSolFree\_SPBCGS

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an 'F' to the function name.

The SUNLINSOL\_SPBCGS module also defines the following additional user-callable functions.

#### |SUNLinSol\_SPBCGSSetPrecType

Call	retval =	SUNLinSol	_SPBCGSSetPre	cTvpe(LS.	pretype):

Description The function SUNLinSol\_SPBCGSSetPrecType updates the type of preconditioning

to use in the SUNLINSOL\_SPBCGS object.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SPBCGS object to update

pretype (int) flag indicating the desired type of preconditioning, allowed values

match those discussed in SUNLinSol\_SPBCGS.

Return value This routine will return with one of the error codes SUNLS\_ILL\_INPUT (illegal

pretype), SUNLS\_MEM\_NULL (S is NULL) or SUNLS\_SUCCESS.

Deprecated Name For backward compatibility, the wrapper function SUNSPBCGSSetPrecType with

idential input and output arguments is also provided.

F2003 Name FSUNLinSol\_SPBCGSSetPrecType

#### SUNLinSol\_SPBCGSSetMaxl

Call retval = SUNLinSol\_SPBCGSSetMaxl(LS, maxl);

Description The function SUNLinSol\_SPBCGSSetMaxl updates the number of linear solver iter-

ations to allow.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SPBCGS object to update

max1 (int) flag indicating the number of iterations to allow. Values  $\leq 0$  will result

in the default value (5).

Return value This routine will return with one of the error codes SUNLS\_MEM\_NULL (S is NULL) or

 ${\tt SUNLS\_SUCCESS}.$ 

Deprecated Name For backward compatibility, the wrapper function SUNSPBCGSSetMaxl with idential

input and output arguments is also provided.

F2003 Name FSUNLinSol\_SPBCGSSetMaxl

#### SUNLinSolSetInfoFile\_SPBCGS

Call retval = SUNLinSolSetInfoFile\_SPBCGS(LS, info\_file);

Description The function SUNLinSolSetInfoFile\_SPBCGS sets the output file where all informative

(non-error) messages should be directed.

Arguments LS (SUNLinearSolver) a SUNNONLINSOL object

Return value The return value is

• SUNLS\_SUCCESS if successful

- SUNLS\_MEM\_NULL if the SUNLinearSolver memory was NULL
- SUNLS\_ILL\_INPUT if SUNDIALS was not built with monitoring enabled

Notes This function is intended for users that wish to monitor the linear solver progress. By default, the file pointer is set to stdout.

SUNDIALS must be built with the CMake option SUNDIALS\_BUILD\_WITH\_MONITORING, to utilize this function. See section A.1.2 for more information.

F2003 Name FSUNLinSolSetInfoFile\_SPBCGS

#### SUNLinSolSetPrintLevel\_SPBCGS

Call retval = SUNLinSolSetPrintLevel\_SPBCGS(NLS, print\_level);

Description The function SUNLinSolSetPrintLevel\_SPBCGS specifies the level of verbosity of the

output.

Arguments LS (SUNLinearSolver) a SUNNONLINSOL object

print\_level (int) flag indicating level of verbosity; must be one of:

- 0, no information is printed (default)
- 1, for each linear iteration the residual norm is printed

Return value The return value is

- SUNLS\_SUCCESS if successful
- SUNLS\_MEM\_NULL if the SUNLinearSolver memory was NULL
- SUNLS\_ILL\_INPUT if SUNDIALS was not built with monitoring enabled, or the print level value was invalid

Notes This function is intended for users that wish to monitor the linear solver progress. By default, the print level is 0.

SUNDIALS must be built with the CMake option SUNDIALS\_BUILD\_WITH\_MONITORING, to utilize this function. See section A.1.2 for more information.

F2003 Name FSUNLinSolSetPrintLevel\_SPBCGS

#### 9.16.3 SUNLinearSolver\_SPBCGS Fortran interfaces

The SUNLINSOL\_SPBCGS module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

#### FORTRAN 2003 interface module

The fsunlinsol\_spbcgs\_mod FORTRAN module defines interfaces to all SUNLINSOL\_SPBCGS C functions using the intrinsic iso\_c\_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function SUNLinSol\_SPBCGS is interfaced as FSUNLinSol\_SPBCGS.

The Fortran 2003 sunlinsol\_spbcgs interface module can be accessed with the use statement, i.e. use fsunlinsol\_spbcgs\_mod, and linking to the library libsundials\_fsunlinsolspbcgs\_mod. lib in addition to the C library. For details on where the library and module file

fsunlinsol\_spbcgs\_mod.mod are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the libsundials\_fsunlinsolspbcgs\_mod library.

#### FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL\_SPBCGS module also includes a Fortran-callable function for creating a SUNLinearSolver object.

#### FSUNSPBCGSINIT

Call FSUNSPBCGSINIT(code, pretype, maxl, ier)

Description The function FSUNSPBCGSINIT can be called for Fortran programs to create a SUNLIN-SOL\_SPBCGS object.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3

for KINSOL, and 4 for ARKODE).

pretype (int\*) flag indicating desired preconditioning type
maxl (int\*) flag indicating number of iterations to allow

Return value ier is a return completion flag equal to 0 for a success return and -1 otherwise. See

printed message for details in case of failure.

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

Allowable values for pretype and max1 are the same as for the C function

SUNLinSol\_SPBCGS.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL\_SPBCGS module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

#### FSUNMASSSPBCGSINIT

Call FSUNMASSSPBCGSINIT(pretype, maxl, ier)

Description The function FSUNMASSSPBCGSINIT can be called for Fortran programs to create a SUN-

LINSOL\_SPBCGS object for mass matrix linear systems.

Arguments pretype (int\*) flag indicating desired preconditioning type

maxl (int\*) flag indicating number of iterations to allow

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

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

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

 ${\tt SUNLinSol\_SPBCGS}.$ 

The SUNLinSol\_SPBCGSSetPrecType and SUNLinSol\_SPBCGSSetMaxl routines also support Fortran interfaces for the system and mass matrix solvers.

#### FSUNSPBCGSSETPRECTYPE

Call FSUNSPBCGSSETPRECTYPE(code, pretype, ier)

Description The function FSUNSPBCGSSETPRECTYPE can be called for Fortran programs to change

the type of preconditioning to use.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3

for KINSOL, and 4 for ARKODE).

pretype (int\*) flag indicating the type of preconditioning to use.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPBCGSSetPrecType for complete further documentation of this routine.

#### FSUNMASSSPBCGSSETPRECTYPE

Call FSUNMASSSPBCGSSETPRECTYPE(pretype, ier)

Description The function FSUNMASSSPBCGSSETPRECTYPE can be called for Fortran programs to change

the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to FSUNSPBCGSSETPRECTYPE above, except that code is not

needed since mass matrix linear systems only arise in ARKODE.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPBCGSSetPrecType for complete further documentation of this routine.

#### FSUNSPBCGSSETMAXL

Call FSUNSPBCGSSETMAXL(code, maxl, ier)

Description The function FSUNSPBCGSSETMAXL can be called for Fortran programs to change the

maximum number of iterations to allow.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for

KINSOL, and 4 for ARKODE).

maxl (int\*) the number of iterations to allow.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPBCGSSetMaxl for complete further documentation of this routine.

#### FSUNMASSSPBCGSSETMAXL

Call FSUNMASSSPBCGSSETMAXL(maxl, ier)

Description The function FSUNMASSSPBCGSSETMAXL can be called for Fortran programs to change

the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to FSUNSPBCGSSETMAXL above, except that code is not

needed since mass matrix linear systems only arise in ARKODE.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPBCGSSetMaxl for complete further documentation of this routine.

#### 9.16.4 SUNLinearSolver\_SPBCGS content

The SUNLINSOL\_SPBCGS module defines the *content* field of a SUNLinearSolver as the following structure:

```
struct _SUNLinearSolverContent_SPBCGS {
  int maxl;
  int pretype;
  int numiters;
  realtype resnorm;
  int last_flag;
  ATimesFn ATimes;
  void* ATData;
  PSetupFn Psetup;
  PSolveFn Psolve;
  void* PData;
  N_Vector s1;
  N_Vector s2;
  N_Vector r;
  N_Vector r_star;
  N_Vector p;
  N_Vector q;
  N_Vector u;
  N_Vector Ap;
  N_Vector vtemp;
            print_level;
  int
  FILE*
            info_file;
};
These entries of the content field contain the following information:
             - number of SPBCGS iterations to allow (default is 5),
maxl
             - flag for type of preconditioning to employ (default is none),
pretype
             - number of iterations from the most-recent solve,
numiters
             - final linear residual norm from the most-recent solve,
resnorm
             - last error return flag from an internal function,
last_flag
             - function pointer to perform Av product,
ATimes
             - pointer to structure for ATimes,
ATData
             - function pointer to preconditioner setup routine,
Psetup
Psolve
             - function pointer to preconditioner solve routine,
PData
             - pointer to structure for Psetup and Psolve,
             - vector pointers for supplied scaling matrices (default is NULL),
s1, s2
             - a NVECTOR which holds the current scaled, preconditioned linear system residual,
r_star
             - a NVECTOR which holds the initial scaled, preconditioned linear system residual,
p, q, u, Ap, vtemp - NVECTORS used for workspace by the SPBCGS algorithm.
print_level - controls the amount of information to be printed to the info file
             - the file where all informative (non-error) messages will be directed
info_file
```

## 9.17 The SUNLinearSolver\_SPTFQMR implementation

This section describes the SUNLINSOL implementation of the SPTFQMR (Scaled, Preconditioned, Transpose-Free Quasi-Minimum Residual [27]) iterative linear solver. The SUNLINSOL\_SPTFQMR module is designed to be compatible with any NVECTOR implementation that supports a minimal subset of operations (N\_VClone, N\_VDotProd, N\_VScale, N\_VLinearSum, N\_VProd, N\_VConst, N\_VDiv, and N\_VDestroy). Unlike the SPGMR and SPFGMR algorithms, SPTFQMR requires a fixed amount of memory that does not increase with the number of allowed iterations.

To access the SUNLINSOL\_SPTFQMR module, include the header file sunlinsol/sunlinsol\_sptfqmr.h. We note that the SUNLINSOL\_SPTFQMR module is accessible from SUNDIALS packages without separately linking to the libsundials\_sunlinsolsptfqmr module library.

#### 9.17.1 SUNLinearSolver\_SPTFQMR description

This solver is constructed to perform the following operations:

- During construction all NVECTOR solver data is allocated, with vectors cloned from a template NVECTOR that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLINSOL\_SPTFQMR to supply the ATimes, PSetup, and Psolve function pointers and s1 and s2 scaling vectors.
- In the "initialize" call, the solver parameters are checked for validity.
- In the "setup" call, any non-NULL PSetup function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic PSetup function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call the TFQMR iteration is performed. This will include scaling and preconditioning if those options have been supplied.

#### 9.17.2 SUNLinearSolver\_SPTFQMR functions

The SUNLINSOL\_SPTFQMR module provides the following user-callable constructor for creating a SUNLinearSolver object.

# SUNLinSol\_SPTFQMR

Call LS = SUNLinSol\_SPTFQMR(y, pretype, maxl);

Description The function SUNLinSol\_SPTFQMR creates and allocates memory for a SPTFQMR

SUNLinearSolver object.

Arguments y (N\_Vector) a template for cloning vectors needed within the solver pretype (int) flag indicating the desired type of preconditioning, allowed values are:

- PREC\_NONE (0)
- PREC\_LEFT (1)
- PREC\_RIGHT (2)
- PREC\_BOTH (3)

Any other integer input will result in the default (no preconditioning). (int) the number of linear iterations to allow. Values  $\leq 0$  will result in the default value (5).

maxl

Return value This returns a SUNLinearSolver object. If either y is incompatible then this

routine will return NULL.

Notes This routine will perform consistency checks to ensure that it is called with a consistent NVECTOR implementation (i.e. that it supplies the requisite vector operations).

If y is incompatible, then this routine will return NULL.

We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL). While it is possible to configure a SUNLINSOL\_SPTFQMR object to use any of the preconditioning options with these solvers, this use mode is not supported and may result

in inferior performance.

Deprecated Name For backward compatibility, the wrapper function SUNSPTFQMR with idential input

and output arguments is also provided.

F2003 Name FSUNLinSol\_SPTFQMR

The SUNLINSOL\_SPTFQMR module defines implementations of all "iterative" linear solver operations listed in Sections 9.1.1 - 9.1.3:

• SUNLinSolGetType\_SPTFQMR

- SUNLinSolInitialize\_SPTFQMR
- SUNLinSolSetATimes\_SPTFQMR
- SUNLinSolSetPreconditioner\_SPTFQMR
- SUNLinSolSetScalingVectors\_SPTFQMR
- SUNLinSolSetup\_SPTFQMR
- SUNLinSolSolve\_SPTFQMR
- SUNLinSolNumIters\_SPTFQMR
- SUNLinSolResNorm\_SPTFQMR
- SUNLinSolResid\_SPTFQMR
- SUNLinSolLastFlag\_SPTFQMR
- SUNLinSolSpace\_SPTFQMR
- SUNLinSolFree\_SPTFQMR

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an 'F' to the function name.

The SUNLINSOL\_SPTFQMR module also defines the following additional user-callable functions.

#### SUNLinSol\_SPTFQMRSetPrecType

Call retval = SUNLinSol\_SPTFQMRSetPrecType(LS, pretype);

to use in the SUNLINSOL\_SPTFQMR object.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SPTFQMR object to update

pretype (int) flag indicating the desired type of preconditioning, allowed values

match those discussed in SUNLinSol\_SPTFQMR.

Return value This routine will return with one of the error codes SUNLS\_ILL\_INPUT (illegal

pretype), SUNLS\_MEM\_NULL (S is NULL) or SUNLS\_SUCCESS.

 $Deprecated\ Name\ For\ backward\ compatibility,\ the\ wrapper\ function\ {\tt SUNSPTFQMRSetPrecType}\ with$ 

idential input and output arguments is also provided.

F2003 Name FSUNLinSol\_SPTFQMRSetPrecType

#### SUNLinSol\_SPTFQMRSetMaxl

Call retval = SUNLinSol\_SPTFQMRSetMaxl(LS, maxl);

Description The function SUNLinSol\_SPTFQMRSetMaxl updates the number of linear solver iterations

to allow.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_SPTFQMR object to update

maxl (int) flag indicating the number of iterations to allow; values  $\leq 0$  will result in

the default value (5)

Return value This routine will return with one of the error codes SUNLS\_MEM\_NULL (S is NULL) or

SUNLS\_SUCCESS.

F2003 Name FSUNLinSol\_SPTFQMRSetMaxl

 ${\bf SUNSPTFQMRSetMaxl}$ 

#### SUNLinSolSetInfoFile\_SPTFQMR

Call retval = SUNLinSolSetInfoFile\_SPTFQMR(LS, info\_file);

Description The function SUNLinSolSetInfoFile\_SPTFQMR sets the output file where all informative

(non-error) messages should be directed.

Arguments LS (SUNLinearSolver) a SUNNONLINSOL object

info\_file (FILE\*) pointer to output file (stdout by default); a NULL input will disable

output

Return value The return value is

• SUNLS\_SUCCESS if successful

• SUNLS\_MEM\_NULL if the SUNLinearSolver memory was NULL

• SUNLS\_ILL\_INPUT if SUNDIALS was not built with monitoring enabled

Notes This function is intended for users that wish to monitor the linear solver progress. By default, the file pointer is set to stdout.

SUNDIALS must be built with the CMake option SUNDIALS\_BUILD\_WITH\_MONITORING, to utilize this function. See section A.1.2 for more information.

F2003 Name FSUNLinSolSetInfoFile\_SPTFQMR

#### ${\tt SUNLinSolSetPrintLevel\_SPTFQMR}$

Call retval = SUNLinSolSetPrintLevel\_SPTFQMR(NLS, print\_level);

Description The function SUNLinSolSetPrintLevel\_SPTFQMR specifies the level of verbosity of the

output.

Arguments LS (SUNLinearSolver) a SUNNONLINSOL object

print\_level (int) flag indicating level of verbosity; must be one of:

- 0, no information is printed (default)
- 1, for each linear iteration the residual norm is printed

Return value The return value is

- SUNLS\_SUCCESS if successful
- SUNLS\_MEM\_NULL if the SUNLinearSolver memory was NULL
- SUNLS\_ILL\_INPUT if SUNDIALS was not built with monitoring enabled, or the print level value was invalid

Notes This function is intended for users that wish to monitor the linear solver progress. By

default, the print level is 0.

SUNDIALS must be built with the CMake option SUNDIALS\_BUILD\_WITH\_MONITORING,

to utilize this function. See section A.1.2 for more information.

F2003 Name FSUNLinSolSetPrintLevel\_SPTFQMR

#### 9.17.3SUNLinearSolver\_SPTFQMR Fortran interfaces

The SUNLINSOL\_SPFGMR module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

#### FORTRAN 2003 interface module

The fsunlinsol\_sptfqmr\_mod FORTRAN module defines interfaces to all SUNLINSOL\_SPFGMR C functions using the intrinsic iso\_c\_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function SUNLinSol\_SPTFQMR is interfaced as FSUNLinSol\_SPTFQMR.

The FORTRAN 2003 SUNLINSOL\_SPFGMR interface module can be accessed with the use statement, i.e. use fsunlinsol\_sptfqmr\_mod, and linking to the library libsundials\_fsunlinsolsptfqmr\_mod. lib in addition to the C library. For details on where the library and module file

fsunlinsol\_sptfqmr\_mod.mod are installed see Appendix A. We note that the module is accessible from the Fortran 2003 sundials integrators without separately linking to the libsundials\_fsunlinsolsptfqmr\_mod library.

#### FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL\_SPTFQMR module also includes a Fortran-callable function for creating a SUNLinearSolver object.

#### FSUNSPTFQMRINIT

Call FSUNSPTFQMRINIT(code, pretype, maxl, ier)

Description The function FSUNSPTFQMRINIT can be called for Fortran programs to create a SUNLIN-

SOL\_SPTFQMR object.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3

for KINSOL, and 4 for ARKODE).

pretype (int\*) flag indicating desired preconditioning type

(int\*) flag indicating number of iterations to allow maxl

Return value ier is a return completion flag equal to 0 for a success return and -1 otherwise. See

printed message for details in case of failure.

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

Allowable values for pretype and max1 are the same as for the C function

SUNLinSol\_SPTFQMR.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL\_SPTFQMR module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

#### FSUNMASSSPTFQMRINIT

Call FSUNMASSSPTFQMRINIT(pretype, maxl, ier)

Description The function FSUNMASSSPTFQMRINIT can be called for Fortran programs to create a

SUNLINSOL\_SPTFQMR object for mass matrix linear systems.

Arguments pretype (int\*) flag indicating desired preconditioning type

maxl (int\*) flag indicating number of iterations to allow

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

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

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

SUNLinSol\_SPTFQMR.

The SUNLinSol\_SPTFQMRSetPrecType and SUNLinSol\_SPTFQMRSetMaxl routines also support Fortran interfaces for the system and mass matrix solvers.

#### FSUNSPTFQMRSETPRECTYPE

Call FSUNSPTFQMRSETPRECTYPE(code, pretype, ier)

Description The function FSUNSPTFQMRSETPRECTYPE can be called for Fortran programs to change

the type of preconditioning to use.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3

for KINSOL, and 4 for ARKODE).

pretype (int\*) flag indicating the type of preconditioning to use.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPTFQMRSetPrecType for complete further documentation of this rou-

tine.

#### FSUNMASSSPTFQMRSETPRECTYPE

Call FSUNMASSSPTFQMRSETPRECTYPE(pretype, ier)

Description The function FSUNMASSSPTFQMRSETPRECTYPE can be called for Fortran programs to

change the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to FSUNSPTFQMRSETPRECTYPE above, except that code is

not needed since mass matrix linear systems only arise in ARKODE.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPTFQMRSetPrecType for complete further documentation of this rou-

tine.

#### FSUNSPTFQMRSETMAXL

Call FSUNSPTFQMRSETMAXL(code, maxl, ier)

Description The function FSUNSPTFQMRSETMAXL can be called for Fortran programs to change the

maximum number of iterations to allow.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for

KINSOL, and 4 for ARKODE).

maxl (int\*) the number of iterations to allow.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPTFQMRSetMaxl for complete further documentation of this routine.

#### FSUNMASSSPTFQMRSETMAXL

Call FSUNMASSSPTFQMRSETMAXL(maxl, ier)

Description The function FSUNMASSSPTFQMRSETMAXL can be called for Fortran programs to change

the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to FSUNSPTFQMRSETMAXL above, except that code is not

needed since mass matrix linear systems only arise in ARKODE.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_SPTFQMRSetMaxl for complete further documentation of this routine.

#### 9.17.4 SUNLinearSolver\_SPTFQMR content

struct \_SUNLinearSolverContent\_SPTFQMR {

The SUNLINSOL\_SPTFQMR module defines the *content* field of a SUNLinearSolver as the following structure:

```
int maxl;
  int pretype;
  int numiters;
  realtype resnorm;
  int last_flag;
  ATimesFn ATimes;
  void* ATData;
  PSetupFn Psetup;
  PSolveFn Psolve;
  void* PData;
  N_Vector s1;
  N_Vector s2;
  N_Vector r_star;
  N_Vector q;
  N_Vector d;
  N_Vector v;
  N_Vector p;
  N_Vector *r;
  N_Vector u;
  N_Vector vtemp1;
  N_Vector vtemp2;
  N_Vector vtemp3;
  int
            print_level;
  FILE*
            info_file;
};
These entries of the content field contain the following information:
             - number of TFQMR iterations to allow (default is 5),
maxl
             - flag for type of preconditioning to employ (default is none),
pretype
             - number of iterations from the most-recent solve,
numiters
             - final linear residual norm from the most-recent solve,
resnorm
             - last error return flag from an internal function,
last_flag
             - function pointer to perform Av product,
ATimes
ATData
             - pointer to structure for ATimes,
             - function pointer to preconditioner setup routine,
Psetup
```

Psolve - function pointer to preconditioner solve routine,

PData - pointer to structure for Psetup and Psolve,

s1, s2 - vector pointers for supplied scaling matrices (default is NULL),

r\_star - a NVECTOR which holds the initial scaled, preconditioned linear system residual,

q, d, v, p, u - NVECTORS used for workspace by the SPTFQMR algorithm,

- array of two NVECTORS used for workspace within the SPTFQMR algorithm,

vtemp1, vtemp2, vtemp3 - temporary vector storage.

print\_level - controls the amount of information to be printed to the info file

info\_file - the file where all informative (non-error) messages will be directed

## 9.18 The SUNLinearSolver\_PCG implementation

This section describes the SUNLINSOL implementation of the PCG (Preconditioned Conjugate Gradient [29]) iterative linear solver. The SUNLINSOL\_PCG module is designed to be compatible with any NVECTOR implementation that supports a minimal subset of operations (N\_VClone, N\_VDotProd, N\_VScale, N\_VLinearSum, N\_VProd, and N\_VDestroy). Unlike the SPGMR and SPFGMR algorithms, PCG requires a fixed amount of memory that does not increase with the number of allowed iterations.

To access the SUNLINSOL\_PCG module, include the header file sunlinsol\_pcg.h. We note that the SUNLINSOL\_PCG module is accessible from SUNDIALS packages without separately linking to the libsundials\_sunlinsolpcg module library.

## 9.18.1 SUNLinearSolver\_PCG description

Unlike all of the other iterative linear solvers supplied with SUNDIALS, PCG should only be used on symmetric linear systems (e.g. mass matrix linear systems encountered in ARKODE). As a result, the explanation of the role of scaling and preconditioning matrices given in general must be modified in this scenario. The PCG algorithm solves a linear system Ax = b where A is a symmetric  $(A^T = A)$ , real-valued matrix. Preconditioning is allowed, and is applied in a symmetric fashion on both the right and left. Scaling is also allowed and is applied symmetrically. We denote the preconditioner and scaling matrices as follows:

- P is the preconditioner (assumed symmetric),
- $\bullet$  S is a diagonal matrix of scale factors.

The matrices A and P are not required explicitly; only routines that provide A and  $P^{-1}$  as operators are required. The diagonal of the matrix S is held in a single NVECTOR, supplied by the user.

In this notation, PCG applies the underlying CG algorithm to the equivalent transformed system

$$\tilde{A}\tilde{x} = \tilde{b} \tag{9.4}$$

where

$$\tilde{A} = SP^{-1}AP^{-1}S,$$

$$\tilde{b} = SP^{-1}b,$$

$$\tilde{x} = S^{-1}Px.$$
(9.5)

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:

$$\|\tilde{b} - \tilde{A}\tilde{x}\|_{2} < \delta$$

$$\Leftrightarrow \qquad \|SP^{-1}b - SP^{-1}Ax\|_{2} < \delta$$

$$\Leftrightarrow \qquad \|P^{-1}b - P^{-1}Ax\|_{S} < \delta$$

where  $||v||_S = \sqrt{v^T S^T S v}$ , with an input tolerance  $\delta$ .

This solver is constructed to perform the following operations:

- During construction all NVECTOR solver data is allocated, with vectors cloned from a template NVECTOR that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLINSOL\_PCG to supply the ATimes, PSetup, and Psolve function pointers and s scaling vector.
- In the "initialize" call, the solver parameters are checked for validity.
- In the "setup" call, any non-NULL PSetup function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic PSetup function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call the PCG iteration is performed. This will include scaling and preconditioning if those options have been supplied.

#### 9.18.2SUNLinearSolver\_PCG functions

The SUNLINSOL\_PCG module provides the following user-callable constructor for creating a SUNLinearSolver object.

CITAIT		Daa
SUNI	.inSol	PCG:

Notes

Call LS = SUNLinSol\_PCG(y, pretype, maxl);

Description The function SUNLinSol\_PCG creates and allocates memory for a PCG SUNLinearSolver

object.

Arguments (N\_Vector) a template for cloning vectors needed within the solver

> pretype (int) flag indicating whether to use preconditioning. Since the PCG algorithm is designed to only support symmetric preconditioning, then any of the pretype inputs PREC\_LEFT (1), PREC\_RIGHT (2), or PREC\_BOTH (3) will result in use of the symmetric preconditioner; any other integer input

will result in the default (no preconditioning).

(int) the number of linear iterations to allow; values  $\leq 0$  will result in maxl

the default value (5).

Return value This returns a SUNLinearSolver object. If either y is incompatible then this

routine will return NULL.

This routine will perform consistency checks to ensure that it is called with a consistent NVECTOR implementation (i.e. that it supplies the requisite vector operations). If y is incompatible, then this routine will return NULL.

Although some Sundials solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL), PCG should only be used with these packages when the linear systems are known to be symmetric. Since the scaling of matrix rows and columns must be identical in a symmetric matrix, symmetric preconditioning should work appropriately even for packages designed with one-sided preconditioning in mind.

Deprecated Name For backward compatibility, the wrapper function SUNPCG with idential input and output arguments is also provided.

F2003 Name FSUNLinSol\_PCG

The SUNLINSOL\_PCG module defines implementations of all "iterative" linear solver operations listed in Sections 9.1.1 - 9.1.3:

- SUNLinSolGetType\_PCG
- SUNLinSolInitialize\_PCG
- SUNLinSolSetATimes\_PCG
- SUNLinSolSetPreconditioner\_PCG
- SUNLinSolSetScalingVectors\_PCG since PCG only supports symmetric scaling, the second NVECTOR argument to this function is ignored
- SUNLinSolSetup\_PCG
- SUNLinSolSolve\_PCG
- SUNLinSolNumIters\_PCG
- SUNLinSolResNorm\_PCG
- SUNLinSolResid\_PCG
- SUNLinSolLastFlag\_PCG
- SUNLinSolSpace\_PCG
- SUNLinSolFree\_PCG

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an 'F' to the function name.

The SUNLINSOL\_PCG module also defines the following additional user-callable functions.

## SUNLinSol\_PCGSetPrecType

Call retval = SUNLinSol\_PCGSetPrecType(LS, pretype);

Description The function SUNLinSol\_PCGSetPrecType updates the flag indicating use of pre-

conditioning in the SUNLINSOL\_PCG object.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_PCG object to update

pretype (int) flag indicating use of preconditioning, allowed values match those

discussed in SUNLinSol\_PCG.

Return value This routine will return with one of the error codes SUNLS\_ILL\_INPUT (illegal

pretype), SUNLS\_MEM\_NULL (S is NULL) or SUNLS\_SUCCESS.

Deprecated Name For backward compatibility, the wrapper function SUNPCGSetPrecType with iden-

tial input and output arguments is also provided.

F2003 Name FSUNLinSol\_PCGSetPrecType

#### SUNLinSol\_PCGSetMaxl

Call retval = SUNLinSol\_PCGSetMaxl(LS, maxl);

Description The function SUNLinSol\_PCGSetMaxl updates the number of linear solver iterations

to allow.

Arguments LS (SUNLinearSolver) the SUNLINSOL\_PCG object to update

maxl (int) flag indicating the number of iterations to allow; values  $\leq 0$  will result

in the default value (5)

Return value This routine will return with one of the error codes SUNLS\_MEM\_NULL (S is NULL) or

SUNLS SUCCESS.

Deprecated Name For backward compatibility, the wrapper function SUNPCGSetMax1 with idential

input and output arguments is also provided.

F2003 Name FSUNLinSol\_PCGSetMaxl

#### SUNLinSolSetInfoFile\_PCG

Call retval = SUNLinSolSetInfoFile\_PCG(LS, info\_file);

Description The function SUNLinSolSetInfoFile\_PCG sets the output file where all informative

(non-error) messages should be directed.

Arguments LS (SUNLinearSolver) a SUNNONLINSOL object

info\_file (FILE\*) pointer to output file (stdout by default); a NULL input will disable

output

Return value The return value is

• SUNLS\_SUCCESS if successful

• SUNLS\_MEM\_NULL if the SUNLinearSolver memory was NULL

• SUNLS\_ILL\_INPUT if SUNDIALS was not built with monitoring enabled

Notes This function is intended for users that wish to monitor the linear solver progress. By

default, the file pointer is set to stdout.

SUNDIALS must be built with the CMake option SUNDIALS\_BUILD\_WITH\_MONITORING, to utilize this function. See section A.1.2 for more information.

F2003 Name FSUNLinSolSetInfoFile\_PCG

#### SUNLinSolSetPrintLevel\_PCG

Call retval = SUNLinSolSetPrintLevel\_PCG(NLS, print\_level);

Description The function SUNLinSolSetPrintLevel\_PCG specifies the level of verbosity of the out-

put.

Arguments LS (SUNLinearSolver) a SUNNONLINSOL object

print\_level (int) flag indicating level of verbosity; must be one of:

- 0, no information is printed (default)
- 1, for each linear iteration the residual norm is printed

Return value The return value is

- SUNLS\_SUCCESS if successful
- SUNLS\_MEM\_NULL if the SUNLinearSolver memory was NULL
- SUNLS\_ILL\_INPUT if SUNDIALS was not built with monitoring enabled, or the print level value was invalid

Notes This function is intended for users that wish to monitor the linear solver progress. By

default, the print level is 0.

SUNDIALS must be built with the CMake option SUNDIALS\_BUILD\_WITH\_MONITORING,

to utilize this function. See section A.1.2 for more information.

F2003 Name FSUNLinSolSetPrintLevel\_PCG

#### 9.18.3 SUNLinearSolver\_PCG Fortran interfaces

The SUNLINSOL\_PCG module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

#### FORTRAN 2003 interface module

The fsunlinsol\_pcg\_mod FORTRAN module defines interfaces to all SUNLINSOL\_PCG C functions using the intrinsic iso\_c\_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function SUNLinSol\_PCG is interfaced as FSUNLinSol\_PCG.

The FORTRAN 2003 SUNLINSOL\_PCG interface module can be accessed with the use statement, i.e. use fsunlinsol\_pcg\_mod, and linking to the library libsundials\_fsunlinsolpcg\_mod.lib in addition to the C library. For details on where the library and module file fsunlinsol\_pcg\_mod.mod are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the libsundials\_fsunlinsolpcg\_mod library.

#### FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL\_PCG module also includes a Fortran-callable function for creating a SUNLinearSolver object.

#### FSUNPCGINIT

Call FSUNPCGINIT(code, pretype, maxl, ier)

Description The function FSUNPCGINIT can be called for Fortran programs to create a SUNLIN-

SOL\_PCG object.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3

for KINSOL, and 4 for ARKODE).

 ${\tt pretype}$  (int\*) flag indicating desired preconditioning type

maxl (int\*) flag indicating number of iterations to allow

Return value ier is a return completion flag equal to 0 for a success return and -1 otherwise. See

printed message for details in case of failure.

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

Allowable values for pretype and max1 are the same as for the C function SUNLinSol\_PCG.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL\_PCG module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

#### FSUNMASSPCGINIT

Call FSUNMASSPCGINIT(pretype, maxl, ier)

Description The function FSUNMASSPCGINIT can be called for Fortran programs to create a SUNLIN-

SOL\_PCG object for mass matrix linear systems.

Arguments pretype (int\*) flag indicating desired preconditioning type

maxl (int\*) flag indicating number of iterations to allow

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

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

Allowable values for pretype and maxl are the same as for the C function SUNLinSol\_PCG.

The SUNLinSol\_PCGSetPrecType and SUNLinSol\_PCGSetMaxl routines also support Fortran interfaces for the system and mass matrix solvers.

#### FSUNPCGSETPRECTYPE

Call FSUNPCGSETPRECTYPE(code, pretype, ier)

 $\label{thm:constraint} \textbf{Description} \quad \textbf{The function } \textbf{FSUNPCGSETPRECTYPE} \ \ \textbf{can be called for Fortran programs to change the} \\$ 

type of preconditioning to use.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3

for KINSOL, and 4 for ARKODE).

pretype (int\*) flag indicating the type of preconditioning to use.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_PCGSetPrecType for complete further documentation of this routine.

#### FSUNMASSPCGSETPRECTYPE

Call FSUNMASSPCGSETPRECTYPE(pretype, ier)

Description The function FSUNMASSPCGSETPRECTYPE can be called for Fortran programs to change

the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to FSUNPCGSETPRECTYPE above, except that code is not

needed since mass matrix linear systems only arise in ARKODE.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_PCGSetPrecType for complete further documentation of this routine.

#### FSUNPCGSETMAXL

Call FSUNPCGSETMAXL(code, maxl, ier)

Description The function FSUNPCGSETMAXL can be called for Fortran programs to change the maxi-

mum number of iterations to allow.

Arguments code (int\*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for

KINSOL, and 4 for ARKODE).

maxl (int\*) the number of iterations to allow.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol\_PCGSetMaxl for complete further documentation of this routine.

#### FSUNMASSPCGSETMAXL

Call FSUNMASSPCGSETMAXL(maxl, ier)

Description The function FSUNMASSPCGSETMAXL can be called for Fortran programs to change the

type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to FSUNPCGSETMAXL above, except that code is not needed

since mass matrix linear systems only arise in ARKODE.

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

Notes See SUNLinSol\_PCGSetMaxl for complete further documentation of this routine.

#### 9.18.4 SUNLinearSolver\_PCG content

The SUNLINSOL\_PCG module defines the content field of a SUNLinearSolver as the following structure:

```
struct _SUNLinearSolverContent_PCG {
  int maxl;
  int pretype;
  int numiters;
  realtype resnorm;
  int last_flag;
  ATimesFn ATimes;
  void* ATData:
  PSetupFn Psetup;
  PSolveFn Psolve;
  void* PData;
  N_Vector s;
  N_Vector r;
  N_Vector p;
  N_Vector z;
  N_Vector Ap;
  int
            print_level;
  FILE*
            info_file;
};
These entries of the content field contain the following information:
maxl
             - number of PCG iterations to allow (default is 5),
             - flag for use of preconditioning (default is none),
pretype
             - number of iterations from the most-recent solve,
numiters
             - final linear residual norm from the most-recent solve,
resnorm
             - last error return flag from an internal function,
last_flag
             - function pointer to perform Av product,
ATimes
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,
             - vector pointer for supplied scaling matrix (default is NULL),
             - a NVECTOR which holds the preconditioned linear system residual,
             - NVECTORS used for workspace by the PCG algorithm.
p, z, Ap
print_level - controls the amount of information to be printed to the info file
             - the file where all informative (non-error) messages will be directed
```

## 9.19 SUNLinearSolver Examples

There are SUNLinearSolver examples that may be installed for each implementation; these make use of the functions in test\_sunlinsol.c. These example functions show simple usage of the

SUNLinearSolver family of functions. The inputs to the examples depend on the linear solver type, and are output to stdout if the example is run without the appropriate number of command-line arguments.

The following is a list of the example functions in test\_sunlinsol.c:

- Test\_SUNLinSolGetType: Verifies the returned solver type against the value that should be returned.
- Test\_SUNLinSolInitialize: Verifies that SUNLinSolInitialize can be called and returns successfully.
- Test\_SUNLinSolSetup: Verifies that SUNLinSolSetup can be called and returns successfully.
- Test\_SUNLinSolSolve: Given a SUNMATRIX object A, NVECTOR objects x and b (where Ax = b) and a desired solution tolerance tol, this routine clones x into a new vector y, calls SUNLinSolSolve to fill y as the solution to Ay = b (to the input tolerance), verifies that each entry in x and y match to within 10\*tol, and overwrites x with y prior to returning (in case the calling routine would like to investigate further).
- Test\_SUNLinSolSetATimes (iterative solvers only): Verifies that SUNLinSolSetATimes can be called and returns successfully.
- Test\_SUNLinSolSetPreconditioner (iterative solvers only): Verifies that SUNLinSolSetPreconditioner can be called and returns successfully.
- Test\_SUNLinSolSetScalingVectors (iterative solvers only): Verifies that SUNLinSolSetScalingVectors can be called and returns successfully.
- Test\_SUNLinSolLastFlag: Verifies that SUNLinSolLastFlag can be called, and outputs the result to stdout.
- Test\_SUNLinSolNumIters (iterative solvers only): Verifies that SUNLinSolNumIters can be called, and outputs the result to stdout.
- Test\_SUNLinSolResNorm (iterative solvers only): Verifies that SUNLinSolResNorm can be called, and that the result is non-negative.
- Test\_SUNLinSolResid (iterative solvers only): Verifies that SUNLinSolResid can be called.
- Test\_SUNLinSolSpace verifies that SUNLinSolSpace can be called, and outputs the results to stdout.

We'll note that these tests should be performed in a particular order. For either direct or iterative linear solvers, Test\_SUNLinSolInitialize must be called before Test\_SUNLinSolSetup, which must be called before Test\_SUNLinSolSolve. Additionally, for iterative linear solvers

 ${\tt Test\_SUNLinSolSetATimes}, {\tt Test\_SUNLinSolSetPreconditioner} \ {\rm and} \\$ 

Test\_SUNLinSolSetScalingVectors should be called before Test\_SUNLinSolInitialize; similarly Test\_SUNLinSolNumIters, Test\_SUNLinSolResNorm and Test\_SUNLinSolResid should be called after Test\_SUNLinSolSolve. These are called in the appropriate order in all of the example problems.

# Chapter 10

# Description of the SUNNonlinearSolver module

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 section 10.2.

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 10.1 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 sections 10.1.1 - 10.1.3. This is followed by the definition of functions supplied to a nonlinear solver implementation in section 10.1.4. A table of nonlinear solver return codes is given in section 10.1.5. The SunnonlinearSolver type and the generic sunnonlinsol module are defined in section 10.1.6. Section 10.1.7 describes how sunnonlinsol models interface with sundials integrators providing sensitivity analysis capabilities (CVODES and IDAS). Finally, section 10.1.8 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.

#### 10.1 The SUNNonlinear Solver 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 group of functions 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.

#### 10.1.1 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 (SUNNonlinsSolSolve). The remaining three

functions for nonlinear solver initialization (SUNNonlinSolInitialization), setup (SUNNonlinSolSetup), and destruction (SUNNonlinSolFree) are optional.

#### SUNNonlinSolGetType

Call type = SUNNonlinSolGetType(NLS);

Description The required function SUNNonlinSolGetType returns nonlinear solver type.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object.

Return value The return value type (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.

F2003 Name FSUNNonlinSolGetType

#### SUNNonlinSolInitialize

Call retval = SUNNonlinSolInitialize(NLS);

Description The optional function SUNNonlinSolInitialize performs nonlinear solver initialization

and may perform any necessary memory allocations.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object.

Return value The return value retval (of type int) 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 initial-

ization may set this operation to NULL.

F2003 Name FSUNNonlinSolInitialize

#### SUNNonlinSolSetup

Call retval = SUNNonlinSolSetup(NLS, y, mem);

Description The optional function SUNNonlinSolSetup performs any solver setup needed for a non-

linear solve.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object.

y (N\_Vector) the initial iteration passed to the nonlinear solver.

mem (void \*) the SUNDIALS integrator memory structure.

Return value The return value retval (of type int) 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.

F2003 Name FSUNNonlinSolSetup

#### SUNNonlinSolSolve

Call retval = SUNNonlinSolSolve(NLS, y0, ycor, w, tol, callLSetup, mem);

Description The required function SUNNonlinSolSolve solves the nonlinear system F(y) = 0 or

G(y) = y.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object.

y0 (N\_Vector) the predicted value for the new solution state. This must remain unchanged throughout the solution process. See section 10.2 for more

detail on the nonlinear system formulation.

(N\_Vector) on input the initial guess for the correction to the predicted ycor state (zero) and on output the final correction to the predicted state. See section 10.2 for more detail on the nonlinear system formulation.

(N\_Vector) the solution error weight vector used for computing weighted

error norms.

(realtype) the requested solution tolerance in the weighted root-meantol

squared norm.

callLSetup (booleantype) a flag indicating that the integrator recommends for the

linear solver setup function to be called.

(void \*) the SUNDIALS integrator memory structure. mem

Return value The return value retval (of type int) 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 and the integrator should halt and return an error to the user).

F2003 Name FSUNNonlinSolSolve

#### SUNNonlinSolFree

retval = SUNNonlinSolFree(NLS); Call

Description The optional function SUNNonlinSolFree frees any memory allocated by the nonlinear

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object.

Return value The return value retval (of type int) 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.

F2003 Name FSUNNonlinSolFree

#### 10.1.2 SUNNonlinear Solver set functions

The following set 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.

#### SUNNonlinSolSetSysFn

Call retval = SUNNonlinSolSetSysFn(NLS, SysFn);

Description The required function SUNNonlinSolSetSysFn is used to provide the nonlinear solver

> with the function defining the nonlinear system. This is the function F(y) in F(y) = 0for SUNNONLINEARSOLVER\_ROOTFIND modules or G(y) in G(y) = y for

SUNNONLINEARSOLVER\_FIXEDPOINT modules.

(SUNNonlinearSolver) a SUNNONLINSOL object. Arguments NLS

> SysFn (SUNNonlinSolSysFn) the function defining the nonlinear system. See section 10.1.4 for the definition of SUNNonlinSolSysFn.

Return value The return value retval (of type int) should be zero for a successful call, and a negative value for a failure.

F2003 Name FSUNNonlinSolSetSysFn

#### SUNNonlinSolSetLSetupFn

Call retval = SUNNonlinSolSetLSetupFn(NLS, LSetupFn);

Description The optional function SUNNonlinSollSetupFn is called by SUNDIALS integrators to

provide the nonlinear solver with access to its linear solver setup function.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object.

LSetupFn (SUNNonlinSolLSetupFn) a wrapper function to the SUNDIALS integrator's

linear solver setup function. See section 10.1.4 for the definition of

SUNNonlinLSetupFn.

Return value The return value retval (of type int) should be zero for a successful call, and a negative

value for a failure.

Notes The SUNNonlinLSetupFn 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.

 $F2003 \ Name \ FSUNNonlinSolSetLSetupFn$ 

#### SUNNonlinSolSetLSolveFn

Call retval = SUNNonlinSolSetLSolveFn(NLS, LSolveFn);

Description The optional function SUNNonlinSolSetLSolveFn is called by SUNDIALS integrators to

provide the nonlinear solver with access to its linear solver solve function.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object

LSolveFn (SUNNonlinSolLSolveFn) a wrapper function to the SUNDIALS integrator's

linear solver solve function. See section 10.1.4 for the definition of

SUNNonlinSolLSolveFn.

Return value The return value retval (of type int) should be zero for a successful call, and a negative

value for a failure.

Notes The SUNNonlinLSolveFn 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.

F2003 Name FSUNNonlinSolSetLSolveFn

#### SUNNonlinSolSetConvTestFn

Call retval = SUNNonlinSolSetConvTestFn(NLS, CTestFn, ctest\_data);

Description The optional function SUNNonlinSolSetConvTestFn 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 (SUNNonlinearSolver) a SUNNONLINSOL object.

CTestFn (SUNNonlineSolConvTestFn) a SUNDIALS integrator's nonlinear solver con-

vergence test function. See section 10.1.4 for the definition of

SUNNonlinSolConvTestFn.

ctest\_data (void\*) is a data pointer passed to CTestFn every time it is called.

Return value The return value retval (of type int) 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.

F2003 Name FSUNNonlinSolSetConvTestFn

#### SUNNonlinSolSetMaxIters

Call retval = SUNNonlinSolSetMaxIters(NLS, maxiters);

Description The optional function SUNNonlinSolSetMaxIters sets the maximum number of non-

linear solver iterations. This is typically called by SUNDIALS integrators to define their

default iteration limit, but may be adjusted by the user.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object.

maxiters (int) the maximum number of nonlinear iterations.

Return value The return value retval (of type int) should be zero for a successful call, and a negative

value for a failure (e.g., maxiters < 1).

F2003 Name FSUNNonlinSolSetMaxIters

#### 10.1.3 SUNNonlinearSolver get functions

The following get functions allow SUNDIALS integrators to retrieve nonlinear solver statistics. The routines to get number of iterations in the most recent solve (SUNNonlinSolGetNumIters) and number of convergence failures (SUNNonlinSolGetNumConvFails) 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 by the ARKODE and CVODE linear solver interfaces. Otherwise, SUNNonlinSolGetCurIter is optional.

#### SUNNonlinSolGetNumIters

Call retval = SUNNonlinSolGetNumIters(NLS, numiters);

Description The optional function SUNNonlinSolGetNumIters 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 (SUNNonlinearSolver) a SUNNONLINSOL object

numiters (long int\*) the total number of nonlinear solver iterations.

Return value The return value retval (of type int) should be zero for a successful call, and a negative

value for a failure.

F2003 Name FSUNNonlinSolGetNumIters

#### SUNNonlinSolGetCurIter

Call retval = SUNNonlinSolGetCurIter(NLS, iter);

Description The function SUNNonlinSolGetCurIter returns the iteration index of the current non-

linear solve. This function is *required* when using SUNDIALS integrator-provided convergence tests or when using a SUNLINSOL spils linear solver; otherwise it is *optional*.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object

iter (int\*) the nonlinear solver iteration in the current solve starting from zero.

Return value The return value retval (of type int) should be zero for a successful call, and a negative

value for a failure.

 $F2003\ Name\ \texttt{FSUNNonlinSolGetCurIter}$ 

#### SUNNonlinSolGetNumConvFails

Call retval = SUNNonlinSolGetNumConvFails(NLS, nconvfails);

Description The optional function SUNNonlinSolGetNumConvFails 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 (SUNNonlinearSolver) a SUNNONLINSOL object

nconvfails (long int\*) the total number of nonlinear solver convergence failures.

Return value The return value retval (of type int) should be zero for a successful call, and a negative

value for a failure.

F2003 Name FSUNNonlinSolGetNumConvFails

## 10.1.4 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 types for functions provided to a SUNNONLINSOL module are defined in the header file sundials/sundials\_nonlinearsolver.h, and are described below.

#### SUNNonlinSolSysFn

Definition typedef int (\*SUNNonlinSolSysFn)(N\_Vector ycor, N\_Vector F, void\* mem);

Purpose 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. See section 10.2 for more detail on the nonlinear system

formulation.

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 retval (of type int) is zero for a successul solve, a positive value for

a recoverable error, and a negative value for an unrecoverable error.

Notes As discussed in section 10.2, 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 residual. These stored values are then directly used in the integrator-supplied linear solver setup and

solve functions as applicable.

#### SUNNonlinSolLSetupFn

Definition typedef int (\*SUNNonlinSolLSetupFn)(booleantype jbad, booleantype\* jcur, void\* mem);

Purpose 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 retval (of type int) is zero for a successul solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.

Notes

The SUNNonlinLSetupFn 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).

#### SUNNonlinSolLSolveFn

Definition typedef int (\*SUNNonlinSolLSolveFn)(N\_Vector b, void\* mem);

Purpose 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 retval (of type int) is zero for a successul solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.

Notes

The SUNNonlinLSolveFn 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 solve may utilize saved values from the nonlinear system evaluation (e.g., the updated solution).

#### SUNNonlinSolConvTestFn

 $N_Vector\ ewt,\ void*\ ctest_data);$ 

Purpose 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:

 ${\tt SUN\_NLS\_SUCCESS} \qquad \text{the iteration is converged}.$ 

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.

#### 10.1.5 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 below in Table 10.1. Here, negative values correspond to non-recoverable failures, positive values to recoverable failures, and zero to a successful call.

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

Table 10.1: Description of the SUNNonlinearSolver return codes

#### 10.1.6 The generic SUNNonlinear Solver 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;
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

```
int
                           (*free)(SUNNonlinearSolver);
                           (*setsysfn)(SUNNonlinearSolver, SUNNonlinSolSysFn);
  int
  int
                           (*setlsetupfn)(SUNNonlinearSolver, SUNNonlinSolLSetupFn);
  int
                           (*setlsolvefn)(SUNNonlinearSolver, SUNNonlinSolLSolveFn);
  int
                           (*setctestfn)(SUNNonlinearSolver, SUNNonlinSolConvTestFn,
                                         void*);
  int
                           (*setmaxiters)(SUNNonlinearSolver, int);
  int
                           (*getnumiters)(SUNNonlinearSolver, long int*);
                           (*getcuriter)(SUNNonlinearSolver, int*);
  int
  int
                           (*getnumconvfails)(SUNNonlinearSolver, long int*);
};
```

The generic Sunnonlinsol module defines and implements the nonlinear solver operations defined in Sections 10.1.1 – 10.1.3. 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:

The Fortran 2003 interface provides a bind(C) derived-type for the \_generic\_SUNNonlinearSolver and the \_generic\_SUNNonlinearSolver\_Ops structures. Their definition is given below.

```
type, bind(C), public :: SUNNonlinearSolver
type(C_PTR), public :: content
type(C_PTR), public :: ops
end type SUNNonlinearSolver
type, bind(C), public :: SUNNonlinearSolver_Ops
type(C_FUNPTR), public :: gettype
type(C_FUNPTR), public :: initialize
type(C_FUNPTR), public :: setup
type(C_FUNPTR), public :: solve
type(C_FUNPTR), public :: free
type(C_FUNPTR), public :: setsysfn
type(C_FUNPTR), public :: set1setupfn
type(C_FUNPTR), public :: setlsolvefn
type(C_FUNPTR), public :: setctestfn
type(C_FUNPTR), public :: setmaxiters
type(C_FUNPTR), public :: getnumiters
type(C_FUNPTR), public :: getcuriter
type(C_FUNPTR), public :: getnumconvfails
end type SUNNonlinearSolver_Ops
```

## 10.1.7 Usage with sensitivity enabled integrators

When used with SUNDIALS packages that support sensitivity analysis capabilities (e.g., CVODES and IDAS) a special NVECTOR module is used to interface with SUNNONLINSOL modules for solves involving

sensitivity vectors stored in an NVECTOR array. As described below, the NVECTOR\_SENSWRAPPER module is an NVECTOR implementation where the vector content is an NVECTOR array. This wrapper vector allows SUNNONLINSOL modules to operate on data stored as a collection of vectors.

For all SUNDIALS-provided SUNNONLINSOL modules a special constructor wrapper is provided so users do not need to interact directly with the NVECTOR\_SENSWRAPPER module. These constructors follow the naming convention SUNNonlinSol\_\*\*\*Sens(count,...) where \*\*\* is the name of the SUNNONLINSOL module, count is the size of the vector wrapper, and ... are the module-specific constructor arguments.

#### The NVECTOR\_SENSWRAPPER module

This section describes the NVECTOR\_SENSWRAPPER implementation of an NVECTOR. To access the NVECTOR\_SENSWRAPPER module, include the header file sundials/sundials\_nvector\_senswrapper.h.

The NVECTOR\_SENSWRAPPER module defines an N\_Vector implementing all of the standard vectors operations defined in Table 7.1.1 but with some changes to how operations are computed in order to accommodate operating on a collection of vectors.

1. Element-wise vector operations are computed on a vector-by-vector basis. For example, the linear sum of two wrappers containing  $n_v$  vectors of length n, N\_VLinearSum(a,x,b,y,z), is computed as

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

2. The dot product of two wrappers containing  $n_v$  vectors of length n is computed as if it were the dot product of two vectors of length  $nn_v$ . Thus  $d = N_V Dot Prod(x,y)$  is

$$d = \sum_{j=0}^{n_v - 1} \sum_{i=0}^{n-1} x_{j,i} y_{j,i}.$$

3. All norms are computed as the maximum of the individual norms of the  $n_v$  vectors in the wrapper. For example, the weighted root mean square norm  $m = N_V \text{WrmsNorm}(x, w)$  is

$$m = \max_{j} \sqrt{\left(\frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i} w_{j,i})^2\right)}$$

To enable usage alongside other NVECTOR modules the NVECTOR\_SENSWRAPPER functions implementing vector operations have \_SensWrapper appended to the generic vector operation name.

The NVECTOR\_SENSWRAPPER module provides the following constructors for creating an NVECTOR\_SENSWRAPPER:

#### N\_VNewEmpty\_SensWrapper

Call w = N\_VNewEmpty\_SensWrapper(count);

Description The function N\_VNewEmpty\_SensWrapper creates an empty NVECTOR\_SENSWRAPPER wrapper with space for count vectors.

Arguments count (int) the number of vectors the wrapper will contain.

Return value The return value w (of type N\_Vector) will be a NVECTOR object if the constructor exits successfully, otherwise w will be NULL.

F2003 Name FN\_VNewEmpty\_SensWrapper

#### N\_VNew\_SensWrapper

```
Call w = N_VNew_SensWrapper(count, y);
```

Description The function N\_VNew\_SensWrapper creates an NVECTOR\_SENSWRAPPER wrapper con-

taining count vectors cloned from y.

Arguments count (int) the number of vectors the wrapper will contain.

y (N\_Vector) the template vectors to use in creating the vector wrapper.

Return value The return value w (of type N\_Vector) will be a NVECTOR object if the constructor exits successfully, otherwise w will be NULL.

F2003 Name FN\_VNew\_SensWrapper

The NVECTOR\_SENSWRAPPER implementation of the NVECTOR module defines the *content* field of the N\_Vector to be a structure containing an N\_Vector array, the number of vectors in the vector array, and a boolean flag indicating ownership of the vectors in the vector array.

```
struct _N_VectorContent_SensWrapper {
  N_Vector* vecs;
  int nvecs;
  booleantype own_vecs;
};
```

The following macros are provided to access the content of an NVECTOR\_SENSWRAPPER vector.

- NV\_CONTENT\_SW(v) provides access to the content structure
- NV\_VECS\_SW(v) provides access to the vector array
- NV\_NVECS\_SW(v) provides access to the number of vectors
- $\bullet$  NV\_OWN\_VECS\_SW(v) provides access to the ownership flag
- NV\_VEC\_SW(v,i) provides access to the i-th vector in the vector array

#### 10.1.8 Implementing a Custom SUNNonlinear Solver Module

A SUNNONLINSOL implementation must do the following:

- 1. Specify the content of the SUNNONLINSOL module.
- 2. Define and implement the required nonlinear solver operations defined in Sections 10.1.1 10.1.3. 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.
- 3. Define and implement a user-callable constructor to create a SUNNonlinearSolver object.

Additionally, a SUNNonlinearSolver implementation may do the following:

- 1. 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.
- 2. Provide additional user-callable "get" routines acting on the SUNNonlinearSolver object, e.g., for returning various solve statistics.

To aid in the creation of custom SUNNONLINSOL modules the generic SUNNONLINSOL module provides the utility functions SUNNonlinSolNewEmpty and SUNNonlinsolFreeEmpty. When used in custom SUNNONLINSOL constructors, the function SUNNonlinSolNewEmpty will ease the introduction of any new optional nonlinear solver operations to the SUNNONLINSOL API by ensuring only required operations need to be set.

#### SUNNonlinSolNewEmpty

Call NLS = SUNNonlinSolNewEmpty();

Description The function SUNNonlinSolNewEmpty allocates a new generic SUNNONLINSOL object and

initializes its content pointer and the function pointers in the operations structure to

NULL.

Arguments None

Return value This function returns a SUNNonlinearSolver object. If an error occurs when allocating

the object, then this routine will return NULL.

F2003 Name FSUNNonlinSolNewEmpty

#### SUNNonlinSolFreeEmpty

Call SUNNonlinSolFreeEmpty(NLS);

Description 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  $\mathtt{NULL}$ ,

and, if it is not, it will free it as well.

Arguments NLS (SUNNonlinearSolver)

Return value None

F2003 Name FSUNNonlinSolFreeEmpty

#### 10.2 IDA SUNNonlinearSolver interface

As discussed in Chapter 2 each integration step requires the (approximate) solution of the nonlinear system

$$G(y_n) = F\left(t_n, y_n, h_n^{-1} \sum_{i=0}^{q} \alpha_{n,i} y_{n-i}\right) = 0.$$
(10.1)

Rather than solving this system for the new state  $y_n$  IDA 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

$$G(y_{cor}) = F(t_n, y_{pred} + y_{cor}, \dot{y}_{pred} + \alpha y_{cor}) = 0.$$

$$(10.2)$$

where  $\alpha = h_n^{-1} \alpha_{n,0}$ .

The nonlinear system function provided by IDA to the nonlinear solver module internally updates the current value of the new state and its derivative based on the input correction vector. The updated vectors are used when calling the DAE residual function and when setting up linear solves (e.g., for updating the Jacobian or preconditioner).

IDA 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.

#### IDAGetCurrentCj

Call flag = IDAGetCurrentCj(ida\_mem, &cj);

Description The function IDAGetCurrentCj returns the scalar  $c_i$  which is proportional to the inverse

of the step size ( $\alpha$  in (10.2)).

Arguments ida\_mem (void \*) pointer to the IDA memory block.

cj (realtype) the value of  $c_i$ .

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

F2003 Name FIDAGetCurrentCj

#### IDAGetCurrentY

Call flag = IDAGetCurrentY(ida\_mem, &y);

Description The function IDAGetCurrentY returns the current y vector.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

y  $(N_{\text{-}}Vector *)$  the current y vector

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

F2003 Name FIDAGetCurrentY

#### IDAGetCurrentYp

Call flag = IDAGetCurrentYp(ida\_mem, &yp);

Description The function IDAGetCurrentYp returns the current  $\dot{y}$  vector.

Arguments ida\_mem (void \*) pointer to the IDA memory block.

yp (N\_Vector \*) the current  $\dot{y}$  vector

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

F2003 Name FIDAGetCurrentYp

#### IDAGetNonlinearSystemData

Call flag = IDAGetNonlinearSystemData(ida\_mem, &tcur, &yypred, &yppred, &yyn, &yyn, &res, &cj, user\_data);

Description The function IDAGetNonlinearSystemData returns all internal data required to construct the current nonlinear system (10.2).

Arguments ida\_mem (void \*) pointer to the IDA memory block.

tcur (realtype\*) current value of the independent variable  $t_n$ .

yypred (N\_Vector\*) predicted value of  $y_{pred}$  at  $t_n$ . yppred (N\_Vector\*) predicted value of  $\dot{y}_{pred}$  at  $t_n$ .

yyn (N\_Vector\*) the vector  $y_n$ . This vector may be not current and may need

to be filled (see the note below).

ypn (N\_Vector\*) the vector  $\dot{y}_n$ . This vector may be not current and may need

to be filled (see the note below).

res (N\_Vector\*) the residual function evaluated at the current time and state,

 $F(t_n, y_n, \dot{y}_n)$ . This vector may be not current and may need to be filled (see the note below).

cj (realtype\*) the scalar  $c_j$  which is proportional to the inverse of the step size ( $\alpha$  in (10.2)).

user\_data (void\*\*) pointer to the user-defined data structures

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

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 (see §10.1.4) to an existing SUNNonlinearSolver object (through a call to SUNNonlinSolSetSysFn) or who need access to nonlinear system data to compute the nonlinear system function 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 IDA (through calls to SUNNonlinSolSetLSetupFn and SUNNonlinSolSetLSolveFn respectively) the vectors yyn, 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.,

$$yyn = y_{pred} + y_{cor},$$
  

$$ypn = \dot{y}_{pred} + \alpha \dot{y}_{cor},$$
  

$$res = F(t_n, y_n, \dot{y}_n),$$

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 yyn, ypn, and res are only current when IDAGetNonlinearSystemData is called after an evaluation of the nonlinear system function.

F2003 Name FIDAGetNonlinearSystemData

### IDAComputeY

```
Call flag = IDAComputeY(ida_mem, ycor, y);
```

Description The function computes the current y(t) vector based on the given correction vector from the nonlinear solver.

Arguments ida\_mem (void \*) pointer to the IDA memory block

ycor (N\_Vector) the correction
y (N\_Vector) the output vector

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

F2003 Name FIDAComputeY

#### ${\tt IDAComputeYp}$

```
Call flag = IDAComputeYp(ida_mem, ycor, yp);
```

Description The function computes  $\dot{y}(t)$ .

Arguments ida\_mem (void \*) pointer to the IDA memory block

ycor (N\_Vector) the correction

yp (N\_Vector) the output vector array

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

IDA\_SUCCESS The optional output value has been successfully set.

IDA\_MEM\_NULL The ida\_mem pointer is NULL.

F2003 Name FIDAComputeYp

## 10.3 The SUNNonlinearSolver\_Newton implementation

This section describes the SUNNONLINSOL implementation of Newton's method. To access the SUNNON-LINSOL\_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\_sunnonlinsolnewton module library.

### 10.3.1 SUNNonlinearSolver\_Newton description

To find the solution to

$$F(y) = 0 ag{10.3}$$

given an initial guess  $y^{(0)}$ , Newton's method computes a series of approximate solutions

$$y^{(m+1)} = y^{(m)} + \delta^{(m+1)} \tag{10.4}$$

where m is the Newton iteration index, and the Newton update  $\delta^{(m+1)}$  is the solution of the linear system

$$A(y^{(m)})\delta^{(m+1)} = -F(y^{(m)}), (10.5)$$

in which A is the Jacobian matrix

$$A \equiv \partial F/\partial y \,. \tag{10.6}$$

Depending on the linear solver used, the Sunnonlinsol\_newton module will employ either a Modified Newton method, or an Inexact Newton method [12, 16, 23, 25, 38]. 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  ${\tt SUNNonlinSollSetupFn}$  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,  ${\tt SUNNonlinSollSetupFn}$  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 Chapter 2 for details on this decision.

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.

#### 10.3.2 SUNNonlinearSolver\_Newton functions

The SUNNONLINSOL\_NEWTON module provides the following constructors for creating a SUNNonlinearSolver object.

#### SUNNonlinSol\_Newton

Call NLS = SUNNonlinSol\_Newton(y);

Description The function SUNNonlinSol\_Newton 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 (N\_Vector) a template for cloning vectors needed within the solver.

Return value The return value NLS (of type SUNNonlinearSolver) will be a SUNNONLINSOL object if

the constructor exits successfully, otherwise NLS will be NULL.

F2003 Name FSUNNonlinSol\_Newton

#### SUNNonlinSol\_NewtonSens

Call NLS = SUNNonlinSol\_NewtonSens(count, y);

Description The function SUNNonlinSol\_NewtonSens creates a SUNNonlinearSolver object for use

with SUNDIALS sensitivity enabled integrators (CVODES and IDAS) to solve nonlinear

systems of the form F(y) = 0 using Newton's method.

Arguments count (int) the number of vectors in the nonlinear solve. When integrating a system

containing Ns sensitivities the value of count is:

• Ns+1 if using a *simultaneous* corrector approach.

• Ns if using a *staggered* corrector approach.

y (N\_Vector) a template for cloning vectors needed within the solver.

Return value The return value NLS (of type SUNNonlinearSolver) will be a SUNNONLINSOL object if the constructor exits successfully, otherwise NLS will be NULL.

F2003 Name FSUNNonlinSol\_NewtonSens

The SUNNONLINSOL\_NEWTON module implements all of the functions defined in sections 10.1.1 - 10.1.3 except for the SUNNonlinSolSetup function. 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 sections 10.1.1 - 10.1.3 should be called in favor of the SUNNONLINSOL\_NEWTON-specific implementations.

The SUNNONLINSOL\_NEWTON module also defines the following additional user-callable function.

#### SUNNonlinSolGetSysFn\_Newton

Call retval = SUNNonlinSolGetSysFn\_Newton(NLS, SysFn);

Description The function SUNNonlinSolGetSysFn\_Newton returns the residual function that defines

the nonlinear system.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object

SysFn (SUNNonlinSolSysFn\*) the function defining the nonlinear system.

Return value The return value retval (of type int) 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.

F2003 Name FSUNNonlinSolGetSysFn\_Newton

#### SUNNonlinSolSetInfoFile\_Newton

Call retval = SUNNonlinSolSetInfoFile\_Newton(NLS, info\_file);

tive (non-error) messages should be directed.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object

Return value The return value is

- 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

Notes This function is intended for users that wish to monitor the nonlinear solver progress. By default, the file pointer is set to stdout.

SUNDIALS must be built with the CMake option SUNDIALS\_BUILD\_WITH\_MONITORING, to utilize this function. See section A.1.2 for more information.

F2003 Name FSUNNonlinSolSetInfoFile\_Newton

#### SUNNonlinSolSetPrintLevel\_Newton

Call retval = SUNNonlinSolSetPrintLevel\_Newton(NLS, print\_level);

Description The function SUNNonlinSolSetPrintLevel\_Newton specifies the level of verbosity of

the output.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object

print\_level (int) 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 The return value is

- 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.

SUNDIALS must be built with the CMake option SUNDIALS\_BUILD\_WITH\_MONITORING, to utilize this function. See section A.1.2 for more information.

F2003 Name FSUNNonlinSolSetPrintLevel\_Newton

#### 10.3.3 SUNNonlinearSolver\_Newton Fortran interfaces

The SUNNONLINSOL\_NEWTON module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

#### FORTRAN 2003 interface module

The fsunnonlinsol\_newton\_mod FORTRAN module defines interfaces to all SUNNONLINSOL\_NEWTON 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 SUNNonlinSol\_Newton is interfaced as FSUNNonlinSol\_Newton.

The FORTRAN 2003 SUNNONLINSOL\_NEWTON interface module can be accessed with the use statement, i.e. use fsunnonlinsol\_newton\_mod, and linking to the library

libsundials\_fsunnonlinsolnewton\_mod.lib in addition to the C library. For details on where the library and module file fsunnonlinsol\_newton\_mod.mod are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the libsundials\_fsunnonlinsolnewton\_mod library.

#### FORTRAN 77 interface functions

For SUNDIALS integrators that include a FORTRAN 77 interface, the SUNNONLINSOL\_NEWTON module also includes a Fortran-callable function for creating a SUNNonlinearSolver object.

#### FSUNNEWTONINIT

```
Call FSUNNEWTONINIT(code, ier);
```

Description The function FSUNNEWTONINIT can be called for Fortran programs to create a

SUNNonlinearSolver object for use with SUNDIALS integrators to solve nonlinear sys-

tems of the form F(y) = 0 with Newton's method.

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

for ARKODE).

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

#### 10.3.4 SUNNonlinearSolver\_Newton content

The SUNNONLINSOL\_NEWTON module defines the *content* field of a SUNNonlinearSolver as 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;
  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,

the package-supplied function for setting up the linear solver,
 the package-supplied function for performing a linear solve,
 the function for checking convergence of the Newton iteration.

delta - the Newton iteration update vector,

jcur - the Jacobian status (SUNTRUE = current, SUNFALSE = stale),

- the current number of iterations in the solve attempt,

the maximum number of Newton iterations allowed in a solve,
 the total number of nonlinear iterations across all solves,

nconvfails - the total number of nonlinear convergence failures across all solves, and

ctest\_data - the data pointer passed to the convergence test function.

print\_level - controls the amount of information to be printed to the info fileinfo\_file - the file where all informative (non-error) messages will be directed

## 10.4 The SUNNonlinearSolver\_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 Chapter A.1.4. To access the module, users must include the header file sunnonlinsol\_petscsnes.h. The library to link to is libsundials\_sunnonlinsolpetsc.lib where .lib is .so for shared libaries and .a for static libraries. Users of the SUNNONLINSOL\_PETSCSNES should also see the section NVECTOR\_PETSC 7.8 which discusses the NVECTOR interface to the PETSc Vec API.

#### 10.4.1 SUNNonlinearSolver\_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 [10].

Important usage notes for the SUNNONLINSOL\_PETSCSNES implementation are provided below:

- 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
- 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 SUNNONLINSOL 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.



#### 10.4.2 SUNNonlinearSolver\_PetscSNES functions

The SUNNONLINSOL\_PETSCSNES module provides the following constructor for creating a SUNNonlinearSolver object.

#### SUNNonlinSol\_PetscSNES

Call NLS = SUNNonlinSol\_PetscSNES(y, snes);

Description The function SUNNonlinSol\_PetscSNES creates a SUNNonlinearSolver object that

wraps a PETSc SNES object for use with SUNDIALS. This will call SNESSetFunction on

the provided SNES object.

Arguments snes (SNES) a PETSc SNES object

y (N\_Vector) a N\_Vector object of type NVECTOR\_PETSC that used as a template

for the residual vector

Return value A SUNNONLINSOL object if the constructor exits successfully, otherwise NLS will be NULL.

Notes 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 sections 10.1.1 - 10.1.3 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  $_{\rm PetscSNES}$  appended to the function name. Unless using the SUNNONLINSOL\_PETSCSNES module as a standalone nonlinear solver the generic functions defined in sections  $_{\rm 10.1.1-10.1.3}$  should be called in favor of the SUNNONLINSOL\_PETSCSNES-specific implementations.

The  ${\tt SUNNONLINSOL\_PETSCSNES}$  module also defines the following additional user-callable functions.

#### SUNNonlinSolGetSNES\_PetscSNES

Call retval = SUNNonlinSolGetSNES\_PetscSNES(NLS, SNES\* snes);

Description The function SUNNonlinSolGetSNES\_PetscSNES gets the SNES context that was wrapped.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object

snes (SNES\*) a pointer to a PETSc SNES object that will be set upon return

Return value The return value retval (of type int) should be zero for a successful call, and a negative value for a failure.

#### |SUNNonlinSolGetPetscError\_PetscSNES

Call retval = SUNNonlinSolGetPetscError\_PetscSNES(NLS, PestcErrorCode\* error);

 $\textbf{Description} \quad \textbf{The function $\tt SUNNonlinSolGetPetscError\_PetscSNES} \ gets \ the \ last \ error \ code \ returned$ 

by the last internal call to a PETSc API function.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object

error (PestcErrorCode\*) a pointer to a PETSc error integer that will be set upon

Return value The return value retval (of type int) should be zero for a successful call, and a negative value for a failure.



#### 

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object

SysFn (SUNNonlinSolSysFn\*) the function defining the nonlinear system

Return value The return value retval (of type int) should be zero for a successful call, and a negative value for a failure.

#### 10.4.3 SUNNonlinearSolver\_PetscSNES content

The SUNNONLINSOL\_PETSCSNES module defines the *content* field of a SUNNonlinearSolver as the following structure:

```
struct _SUNNonlinearSolverContent_PetscSNES {
  int sysfn_last_err;
  PetscErrorCode petsc_last_err;
  long int nconvfails;
  long int nni;
  void *imem;
  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 context,
    - the nonlinear residual,
    - wrapper for PETSc vectors used in the system function,
    - wrapper for PETSc vectors used in the system function,
Sys - nonlinear system definining function.
```

# Chapter 11

# Description of the SUNMemory module

To support applications which leverage memory pools, or utilize a memory abstraction layer, SUNDIALS provides a set of utilities we will collectively refer to as the SUNMemoryHelper API. The goal of this API is to allow users to leverage operations defined by native SUNDIALS data structures while allowing the user to have finer-grained control of the memory management.

## 11.1 The SUNMemoryHelper API

This API consists of three new SUNDIALS types: SUNMemoryType, SUNMemory, and SUNMemoryHelper, which we now define.

The SUNMemory structure wraps a pointer to actual data. This structure is defined as

```
typedef struct _SUNMemory
  void*
                ptr;
  SUNMemoryType type;
  booleantype
                own;
} *SUNMemory;
  The SUNMemoryType type is an enumeration that defines the four supported memory types:
typedef enum
  SUNMEMTYPE_HOST,
                         /* pageable memory accessible on the host
                         /* page-locked memory accesible on the host
  SUNMEMTYPE_PINNED,
  SUNMEMTYPE_DEVICE,
                         /* memory accessible from the device
  SUNMEMTYPE_UVM
                         /* memory accessible from the host or device */
} SUNMemoryType;
  Finally, the SUNMemoryHelper structure is defined as
struct _SUNMemoryHelper
{
  void*
                       content;
  SUNMemoryHelper_Ops ops;
} *SUNMemoryHelper;
where SUNMemoryHelper_Ops is defined as
```

```
typedef struct _SUNMemoryHelper_Ops
  /* operations that implementations are required to provide */
                  (*alloc)(SUNMemoryHelper, SUNMemory* memptr, size_t mem_size, SUNMemoryType mem_type
  int
                  (*dealloc)(SUNMemoryHelper, SUNMemory mem);
                  (*copy)(SUNMemoryHelper, SUNMemory dst, SUNMemory src, size_t mem_size);
  int
  /* operations that provide default implementations */
                  (*copyasync)(SUNMemoryHelper, SUNMemory dst, SUNMemory src,
  int
                                size_t mem_size, void* ctx);
  SUNMemoryHelper (*clone)(SUNMemoryHelper);
                  (*destroy) (SUNMemoryHelper);
} *SUNMemoryHelper_Ops;
```

#### 11.1.1 Implementation defined operations

The SUNMemory API also defines the following operations which do require a SUNMemoryHelper instance and **require** the implementation to define them:

```
SUNMemoryHelper_Alloc
```

```
retval = SUNMemoryHelper_Alloc(helper, *memptr, mem_size, mem_type);
Call
Description
             Allocates a SUNMemory object whose ptr field is allocated for mem_size bytes and is
             of type mem_type. The new object will have ownership of ptr and will be deallocated
             when SUNMemoryHelper_Dealloc is called.
Arguments
                      (SUNMemoryHelper) the SUNMemoryHelper object
             helper
             memptr
                       (SUNMemory*) pointer to the allocated SUNMemory
             mem_size (size_t) the size in bytes of the ptr
             mem_type (SUNMemoryType) the SUNMemoryType of the ptr
```

Return value An int flag indicating success (zero) or failure (non-zero).

```
SUNMemoryHelper_Dealloc
```

```
Call
             retval = SUNMemoryHelper_Dealloc(helper, mem);
Description
             Deallocates the mem->ptr field if it is owned by mem, and then deallocates the mem object.
Arguments
             helper (SUNMemoryHelper) the SUNMemoryHelper object
                     (SUNMemory) the SUNMemory object
```

Return value An int flag indicating success (zero) or failure (non-zero).

```
SUNMemoryHelper_Copy
```

```
Call
             retval = SUNMemoryHelper_Copy(helper, dst, src, mem_size);
            Synchronously copies mem_size bytes from the the source memory to the destination
Description
             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.
                       (SUNMemoryHelper) the SUNMemoryHelper object
Arguments
            helper
                       (SUNMemory) the destination memory to copy to
             dst.
```

(SUNMemory) the source memory to copy from src mem\_size (size\_t) the number of bytes to copy

Return value An int flag indicating success (zero) or failure (non-zero).

#### 11.1.2 Utility Functions

The SUNMemoryHelper API defines the following functions which do not require a SUNMemoryHelper instance:

#### SUNMemoryHelper\_Alias

Description Returns a SUNMemory object whose ptr field points to the same address as mem1.

The new object will not have ownership of ptr, therefore, it will not free ptr when

SUNMemoryHelper\_Dealloc is called.

Arguments mem1 (SUNMemory) a SUNMemory object

Return value A SUNMemory object.

#### SUNMemoryHelper\_Wrap

Description Returns a SUNMemory object whose ptr field points to the ptr argument passed to the

function. The new object will not have ownership of ptr, therefore, it will not free ptr

when SUNMemoryHelper\_Dealloc is called.

Arguments ptr (SUNMemoryType) the data pointer to wrap in a SUNMemory object

mem\_type (SUNMemoryType) the SUNMemoryType of the ptr

Return value A SUNMemory object.

#### SUNMemoryHelper\_NewEmpty

Call helper = SUNMemoryHelper\_NewEmpty();

Description Returns an empty SUNMemoryHelper. This is useful for building custom SUNMemoryHelper

implementations.

Arguments

Return value A SUNMemoryHelper object.

#### SUNMemoryHelper\_CopyOps

Call retval = SUNMemoryHelper\_CopyOps(src, dst);

Description Copies the ops field of src to the ops field of dst. This is useful for building custom

SUNMemoryHelper implementations.

Arguments src (SUNMemoryHelper) the object to copy from

dst (SUNMemoryHelper) the object to copy to

Return value An int flag indicating success (zero) or failure (non-zero).

#### 11.1.3 Implementation overridable operations with defaults

In addition, the SUNMemoryHelper API defines the following *optionally overridable* operations which do require a SUNMemoryHelper instance:

#### SUNMemoryHelper\_CopyAsync

Call retval = SUNMemoryHelper\_CopyAsync(helper, dst, src, mem\_size, ctx);

Description Asynchronously copies mem\_size bytes from the the source memory to the destination

memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The helper object should use the memory types of dst and src to determine the appropriate transfer type necessary. The ctx argument is used when a different execution "stream" needs to be provided to perform the copy

in, e.g. with CUDA this would be a cudaStream\_t.

Arguments helper (SUNMemoryHelper) the SUNMemoryHelper object

dst (SUNMemory) the destination memory to copy to src (SUNMemory) the source memory to copy from

mem\_size (size\_t) the number of bytes to copy

ctx (void \*) typically a handle for an object representing an alternate execution

stream, but it can be any implementation specific data

<u>!</u>

Return value An int flag indicating success (zero) or failure (non-zero).

Notes If this operation is not defined by the implementation, then SUNMemoryHelper\_Copy will

be used.

#### SUNMemoryHelper\_Clone

Call helper2 = SUNMemoryHelper\_Clone(helper);

Description Clones the SUNMemoryHelper object itself.

Arguments helper (SUNMemoryHelper) the SUNMemoryHelper object to clone

 $\wedge$ 

Return value A SUNMemoryHelper object.

Notes If this operation is not defined by the implementation, then the default clone will

only copy the SUNMemoryHelper\_Ops structure stored in helper->ops, and not the

helper->content field.

#### SUNMemoryHelper\_Destroy

Call retval = SUNMemoryHelper\_Destroy(helper);

Description Destroys (frees) the SUNMemoryHelper object itself.

Arguments helper (SUNMemoryHelper) the SUNMemoryHelper object to destroy

<u>!\</u>

Return value An int flag indicating success (zero) or failure (non-zero).

Notes If this operation is not defined by the implementation, then the default destroy will only

free the helper->ops field and the helper itself. The helper->content field will not

be freed.

#### 11.1.4 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.

### 11.2 The SUNMemoryHelper\_Cuda implementation

The SUNMemoryHelper\_Cuda module is an implementation of the SUNMemoryHelper API that interfaces to the NVIDIA CUDA [5] library. The implementation defines the constructor

#### SUNMemoryHelper\_Cuda

Call helper = SUNMemoryHelper\_Cuda();

Description Allocates and returns a SUNMemoryHelper object for handling CUDA memory.

Arguments None

Return value A SUNMemoryHelper object if successful, or NULL if not.

#### 11.2.1 SUNMemoryHelper API functions

The implementation provides the following operations defined by the SUNMemoryHelper API:

#### SUNMemoryHelper\_Alloc\_Cuda

Call retval = SUNMemoryHelper\_Alloc\_Cuda(helper, \*memptr, mem\_size, mem\_type);

Description Allocates a SUNMemory object whose ptr field is allocated for mem\_size bytes and is of type mem\_type. The new object will have ownership of ptr and will be deallocated

when SUNMemoryHelper\_Dealloc is called. The SUNMemoryType supported are

• SUNMEMTYPE\_HOST - memory is allocated with a call to malloc

- SUNMEMTYPE\_PINNED memory is allocated with a call to cudaMallocHost
- SUNMEMTYPE\_DEVICE memory is allocated with a call to cudaMalloc
- SUNMEMTYPE\_UVM memory is allocated with a call to cudaMallocManaged

Arguments helper (SUNMemoryHelper) the SUNMemoryHelper object

memptr (SUNMemory\*) pointer to the allocated SUNMemory

mem\_size (size\_t) the size in bytes of the ptr

mem\_type (SUNMemoryType) the SUNMemoryType of the ptr

Return value An int flag indicating success (zero) or failure (non-zero).

#### SUNMemoryHelper\_Dealloc\_Cuda

Call retval = SUNMemoryHelper\_Dealloc\_Cuda(helper, mem);

Description Deallocates the mem->ptr field if it is owned by mem, and then deallocates the mem object.

Arguments helper (SUNMemoryHelper) the SUNMemoryHelper object

mem (SUNMemory) the SUNMemory object

Return value An int flag indicating success (zero) or failure (non-zero).

#### SUNMemoryHelper\_Copy\_Cuda

Call retval = SUNMemoryHelper\_Copy\_Cuda(helper, dst, src, mem\_size);

Description Synchronously copies mem\_size bytes from the the source memory to the destination

memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The helper object will use the memory types of dst

and src to determine the appropriate transfer type necessary.

Arguments This operation uses cudaMemcpy underneath.

```
Return value helper (SUNMemoryHelper) the SUNMemoryHelper object dst (SUNMemory) the destination memory to copy to src (SUNMemory) the source memory to copy from mem_size (size_t) the number of bytes to copy

Notes An int flag indicating success (zero) or failure (non-zero).
```

#### SUNMemoryHelper\_CopyAsync\_Cuda

Call retval = SUNMemoryHelper\_CopyAsync\_Cuda(helper, dst, src, mem\_size, ctx);

Description Asynchronously copies mem\_size bytes from the the source memory to the destination

memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The helper object will use the memory types of dst

and src to determine the appropriate transfer type necessary.

Arguments This operation uses cudaMemcpyAsync underneath.

Return value helper (SUNMemoryHelper) the SUNMemoryHelper object

dst (SUNMemory) the destination memory to copy to src (SUNMemory) the source memory to copy from

mem\_size (size\_t) the number of bytes to copy

ctx (void \*) the cudaStream\_t handle for the stream that the copy will be per-

formed on

Notes An int flag indicating success (zero) or failure (non-zero).

### 11.3 The SUNMemoryHelper\_Hip implementation

The SUNMemoryHelper\_Hip module is an implementation of the SUNMemoryHelper API that interfaces to the AMD ROCm HIP library. The implementation defines the constructor

#### SUNMemoryHelper\_Hip

Call helper = SUNMemoryHelper\_Hip();

Description Allocates and returns a SUNMemoryHelper object for handling HIP memory.

Arguments None

Return value A SUNMemoryHelper object if successful, or NULL if not.

#### 11.3.1 SUNMemoryHelper API functions

The implementation provides the following operations defined by the SUNMemoryHelper API:

#### SUNMemoryHelper\_Alloc\_Hip

```
Call retval = SUNMemoryHelper_Alloc_Hip(helper, *memptr, mem_size, mem_type);
```

Description Allocates a SUNMemory object whose ptr field is allocated for mem\_size bytes and is of type mem\_type. The new object will have ownership of ptr and will be deallocated when SUNMemoryHelper\_Dealloc is called.

The SUNMemoryType supported are

- SUNMEMTYPE\_HOST memory is allocated with a call to malloc
- SUNMEMTYPE\_PINNED memory is allocated with a call to hipMallocHost
- SUNMEMTYPE\_DEVICE memory is allocated with a call to hipMalloc

• SUNMEMTYPE\_UVM - memory is allocated with a call to hipMallocManaged

Arguments helper (SUNMemoryHelper) the SUNMemoryHelper object

memptr (SUNMemory\*) pointer to the allocated SUNMemory

mem\_size (size\_t) the size in bytes of the ptr

mem\_type (SUNMemoryType) the SUNMemoryType of the ptr

Return value An int flag indicating success (zero) or failure (non-zero).

#### SUNMemoryHelper\_Dealloc\_Hip

Call retval = SUNMemoryHelper\_Dealloc\_Hip(helper, mem);

Description Deallocates the mem->ptr field if it is owned by mem, and then deallocates the mem object.

Arguments helper (SUNMemoryHelper) the SUNMemoryHelper object

mem (SUNMemory) the SUNMemory object

Return value An int flag indicating success (zero) or failure (non-zero).

#### SUNMemoryHelper\_Copy\_Hip

Call retval = SUNMemoryHelper\_Copy\_Hip(helper, dst, src, mem\_size);

Description Synchronously copies mem\_size bytes from the the source memory to the destination

memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The helper object will use the memory types of dst

and src to determine the appropriate transfer type necessary.

Arguments This operation uses hipMemcpy underneath.

Return value helper (SUNMemoryHelper) the SUNMemoryHelper object

dst (SUNMemory) the destination memory to copy to src (SUNMemory) the source memory to copy from

mem\_size (size\_t) the number of bytes to copy

Notes An int flag indicating success (zero) or failure (non-zero).

#### SUNMemoryHelper\_CopyAsync\_Hip

Call retval = SUNMemoryHelper\_CopyAsync\_Hip(helper, dst, src, mem\_size, ctx);

Description Asynchronously copies mem\_size bytes from the the source memory to the destination

memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The helper object will use the memory types of dst

and src to determine the appropriate transfer type necessary.

Arguments This operation uses hipMemcpyAsync underneath.

Return value helper (SUNMemoryHelper) the SUNMemoryHelper object

dst (SUNMemory) the destination memory to copy to src (SUNMemory) the source memory to copy from

mem\_size (size\_t) the number of bytes to copy

ctx (void \*) the hipStream\_t handle for the stream that the copy will be per-

formed on

Notes An int flag indicating success (zero) or failure (non-zero).

## 11.4 The SUNMemoryHelper\_Sycl implementation

The SUNMemoryHelper\_Sycl module is an implementation of the SUNMemoryHelper API that interfaces to the SYCL abstraction layer. The implementation defines the constructor

#### SUNMemoryHelper\_Sycl

Call helper = SUNMemoryHelper\_Sycl(Q);

Description Allocates and returns a SUNMemoryHelper object for handling SYCL memory.

Arguments Q (sycl::queue) the queue to use for memory operations

Return value A SUNMemoryHelper object if successful, or NULL if not.

#### 11.4.1 SUNMemoryHelper API functions

The implementation provides the following operations defined by the SUNMemoryHelper API:

#### SUNMemoryHelper\_Alloc\_Sycl

Call retval = SUNMemoryHelper\_Alloc\_Sycl(helper, \*memptr, mem\_size, mem\_type);

Description Allocates a SUNMemory object whose ptr field is allocated for mem\_size bytes and is of type mem\_type. The new object will have ownership of ptr and will be deallocated

when SUNMemoryHelper\_Dealloc is called.

The SUNMemoryType supported are

- SUNMEMTYPE\_HOST memory is allocated with a call to malloc
- SUNMEMTYPE\_PINNED memory is allocated with a call to sycl::malloc\_host
- SUNMEMTYPE\_DEVICE memory is allocated with a call to sycl::malloc\_device
- SUNMEMTYPE\_UVM memory is allocated with a call to sycl::malloc\_shared

Arguments helper (SUNMemoryHelper) the SUNMemoryHelper object

memptr (SUNMemory\*) pointer to the allocated SUNMemory

mem\_size (size\_t) the size in bytes of the ptr

mem\_type (SUNMemoryType) the SUNMemoryType of the ptr

Return value An int flag indicating success (zero) or failure (non-zero).

#### SUNMemoryHelper\_Dealloc\_Sycl

Call retval = SUNMemoryHelper\_Dealloc\_Sycl(helper, mem);

Description Deallocates the mem->ptr field if it is owned by mem, and then deallocates the mem object.

Arguments helper (SUNMemoryHelper) the SUNMemoryHelper object

mem (SUNMemory) the SUNMemory object

Return value An int flag indicating success (zero) or failure (non-zero).

#### SUNMemoryHelper\_Copy\_Sycl

```
Call retval = SUNMemoryHelper_Copy_Sycl(helper, dst, src, mem_size);
```

Description Synchronously copies mem\_size bytes from the the source memory to the destination

memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The helper object will use the memory types of dst

and src to determine the appropriate transfer type necessary.

Arguments This operation uses syclMemcpy underneath.

Return value helper (SUNMemoryHelper) the SUNMemoryHelper object

dst (SUNMemory) the destination memory to copy to src (SUNMemory) the source memory to copy from

mem\_size (size\_t) the number of bytes to copy

Notes An int flag indicating success (zero) or failure (non-zero).

#### |SUNMemoryHelper\_CopyAsync\_Sycl

Call retval = SUNMemoryHelper\_CopyAsync\_Sycl(helper, dst, src, mem\_size, ctx);

Description Asynchronously copies mem\_size bytes from the the source memory to the destination

memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The helper object will use the memory types of dst

and src to determine the appropriate transfer type necessary.

Arguments This operation uses syclMemcpyAsync underneath.

Return value helper (SUNMemoryHelper) the SUNMemoryHelper object

dst (SUNMemory) the destination memory to copy to src (SUNMemory) the source memory to copy from

mem\_size (size\_t) the number of bytes to copy
ctx (void \*) is unsued in this function

Notes An int flag indicating success (zero) or failure (non-zero).

## Appendix A

# SUNDIALS Package Installation Procedure

The installation of any SUNDIALS package is accomplished by installing the SUNDIALS suite as a whole, according to the instructions that follow. The same procedure applies whether or not the downloaded file contains one or all solvers in SUNDIALS.

The SUNDIALS suite (or individual solvers) are distributed as compressed archives (.tar.gz). The name of the distribution archive is of the form *solver-x.y.z.tar.gz*, where *solver* is one of: sundials, cvode, cvodes, arkode, ida, idas, or kinsol, and x.y.z represents the version number (of the SUNDIALS suite or of the individual solver). To begin the installation, first uncompress and expand the sources, by issuing

% tar xzf solver-x.y.z.tar.gz

This will extract source files under a directory *solver*-x.y.z.

Starting with version 2.6.0 of SUNDIALS, CMake is the only supported method of installation. The explanations of the installation procedure begins with a few common observations:

• The remainder of this chapter will follow these conventions:

solverdir is the directory solver-x.y.z created above; i.e., the directory containing the SUNDI-ALS sources.

**builddir** is the (temporary) directory under which SUNDIALS is built.

instdir is the directory under which the SUNDIALS exported header files and libraries will be installed. Typically, header files are exported under a directory instdir/include while libraries are installed under instdir/CMAKE\_INSTALL\_LIBDIR, with instdir and CMAKE\_INSTALL\_LIBDIR specified at configuration time.

- For sundials CMake-based installation, in-source builds are prohibited; in other words, the build directory builddir can **not** be the same as solverdir and such an attempt will lead to an error. This prevents "polluting" the source tree and allows efficient builds for different configurations and/or options.
- The installation directory *instdir* can **not** be the same as the source directory *solverdir*.
- By default, only the libraries and header files are exported to the installation directory *instdir*. If enabled by the user (with the appropriate toggle for CMake), the examples distributed with SUNDIALS will be built together with the solver libraries but the installation step will result in exporting (by default in a subdirectory of the installation directory) the example sources and sample outputs together with automatically generated configuration files that reference the *installed* SUNDIALS headers and libraries. As such, these configuration files for the SUNDIALS examples can be used as "templates" for your own problems. CMake installs CMakeLists.txt files



and also (as an option available only under Unix/Linux) Makefile files. Note this installation approach also allows the option of building the SUNDIALS examples without having to install them. (This can be used as a sanity check for the freshly built libraries.)

• Even if generation of shared libraries is enabled, only static libraries are created for the FCMIX modules. (Because of the use of fixed names for the Fortran user-provided subroutines, FCMIX shared libraries would result in "undefined symbol" errors at link time.)

#### A.1 CMake-based installation

CMake-based installation provides a platform-independent build system. CMake can generate Unix and Linux Makefiles, as well as KDevelop, Visual Studio, and (Apple) XCode project files from the same configuration file. In addition, CMake also provides a GUI front end and which allows an interactive build and installation process.

The SUNDIALS build process requires CMake version 3.1.3 or higher and a working C compiler. On Unix-like operating systems, it also requires Make (and curses, including its development libraries, for the GUI front end to CMake, ccmake), while on Windows it requires Visual Studio. CMake is continually adding new features, and the latest version can be downloaded from http://www.cmake.org. Build instructions for CMake (only necessary for Unix-like systems) can be found on the CMake website. Once CMake is installed, Linux/Unix users will be able to use ccmake, while Windows users will be able to use CMakeSetup.

As previously noted, when using CMake to configure, build and install SUNDIALS, it is always required to use a separate build directory. While in-source builds are possible, they are explicitly prohibited by the SUNDIALS CMake scripts (one of the reasons being that, unlike autotools, CMake does not provide a make distclean procedure and it is therefore difficult to clean-up the source tree after an in-source build). By ensuring a separate build directory, it is an easy task for the user to clean-up all traces of the build by simply removing the build directory. CMake does generate a make clean which will remove files generated by the compiler and linker.

#### A.1.1 Configuring, building, and installing on Unix-like systems

The default CMake configuration will build all included solvers and associated examples and will build static and shared libraries. The *instdir* defaults to /usr/local and can be changed by setting the CMAKE\_INSTALL\_PREFIX variable. Support for FORTRAN and all other options are disabled.

CMake can be used from the command line with the cmake command, or from a curses-based GUI by using the ccmake command. Examples for using both methods will be presented. For the examples shown it is assumed that there is a top level SUNDIALS directory with appropriate source, build and install directories:

```
% mkdir (...)sundials/instdir
% mkdir (...)sundials/builddir
% cd (...)sundials/builddir
```

#### Building with the GUI

Using CMake with the GUI follows this general process:

- Select and modify values, run configure (c key)
- New values are denoted with an asterisk
- To set a variable, move the cursor to the variable and press enter
  - If it is a boolean (ON/OFF) it will toggle the value
  - If it is string or file, it will allow editing of the string

- For file and directories, the <tab> key can be used to complete
- Repeat until all values are set as desired and the generate option is available (g key)
- Some variables (advanced variables) are not visible right away
- To see advanced variables, toggle to advanced mode (t key)
- To search for a variable press / key, and to repeat the search, press the n key

To build the default configuration using the GUI, from the *builddir* enter the ccmake command and point to the *solverdir*:

#### % ccmake ../solverdir

The default configuration screen is shown in Figure A.1.

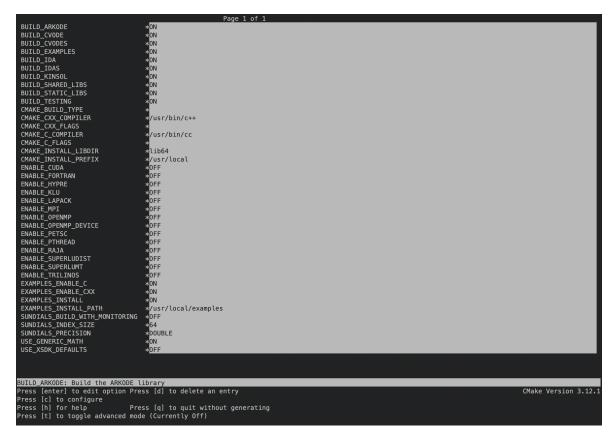


Figure A.1: Default configuration screen. Note: Initial screen is empty. To get this default configuration, press 'c' repeatedly (accepting default values denoted with asterisk) until the 'g' option is available.

The default *instdir* for both SUNDIALS and corresponding examples can be changed by setting the CMAKE\_INSTALL\_PREFIX and the EXAMPLES\_INSTALL\_PATH as shown in figure A.2.

Pressing the (g key) will generate makefiles including all dependencies and all rules to build SUNDIALS on this system. Back at the command prompt, you can now run:

#### % make

To install SUNDIALS in the installation directory specified in the configuration, simply run:

#### % make install

```
BUILD_CVODE
BUILD_CVODES
BUILD EXAMPLES
BUILD_IDA
BUILD_IDAS
BUILD_KINSOL
BUILD_SHARED_LIBS
BUILD_STATIC_LIBS
 CMAKE_CXX_COMPILER
CMAKE_CXX_FLAGS
CMAKE_C_COMPILER
                                                             */usr/bin/c++
                                                               /usr/bin/cc
 CMAKE_C_1 LAGS
CMAKE_INSTALL_LIBDIR
CMAKE_INSTALL_PREFIX
                                                              /usr/casc/sundials/instdir
OFF
 ENABLE CUDA
ENABLE_FORTRAN
ENABLE_HYPRE
ENABLE KLU
 ENABLE_LAPACK
ENABLE_MPI
ENABLE_OPENMP
ENABLE_OPENMP_DEVICE
ENABLE_PETSC
 ENABLE_PTHREAD
ENABLE_RAJA
ENABLE_SUPERLUDIST
ENABLE_SUPERLUMT
ENABLE_TRILINOS
EXAMPLES_ENABLE_C
EXAMPLES_ENABLE_CXX
EXAMPLES_INSTALL
EXAMPLES_INSTALL_PATH
                                                            */usr/casc/sundials/instdir/examples
*OFF
*64
 SUNDIALS_BUILD_WITH_MONITORING
SUNDIALS_INDEX_SIZE
 SUNDIALS PRECISION
                                                               DOUBL F
USE_XSDK_DEFAULTS
EXAMPLES_INSTALL_PATH: Output directory for installing example files
Press [enter] to edit option Press [d] to delete an entry
 ress [c] to configure
ress [h] for help Press [q] to quit without generating
ress [t] to toggle advanced mode (Currently Off)
```

Figure A.2: Changing the *instdir* for SUNDIALS and corresponding examples

#### Building from the command line

Using CMake from the command line is simply a matter of specifying CMake variable settings with the cmake command. The following will build the default configuration:

```
% cmake -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> ../solverdir
% make
% make install
```

### A.1.2 Configuration options (Unix/Linux)

A complete list of all available options for a CMake-based SUNDIALS configuration is 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 (CMAKE\_C\_FLAGS used), Debug, Release, RelWithDebInfo, and MinSizeRel

Default:

Note: Specifying a build type will trigger the corresponding build type specific compiler flag options below which will be appended to the flags set by CMAKE\_<language>\_FLAGS.

CMAKE\_C\_COMPILER - C compiler

Default: /usr/bin/cc

CMAKE\_C\_FLAGS - Flags for C compiler

Default:

CMAKE\_C\_FLAGS\_DEBUG - Flags used by the C compiler during debug builds

Default: -g

CMAKE\_C\_FLAGS\_MINSIZEREL - Flags used by the C compiler during release minsize builds

Default: -Os -DNDEBUG

CMAKE\_C\_FLAGS\_RELEASE - Flags used by the C compiler during release builds

Default: -O3 -DNDEBUG

CMAKE\_CXX\_COMPILER - C++ compiler

Default: /usr/bin/c++

Note: A C++ compiler (and all related options) are only triggered if C++ examples are enabled (EXAMPLES\_ENABLE\_CXX is ON). All SUNDIALS solvers can be used from C++ applications by default without setting any additional configuration options.

CMAKE\_CXX\_FLAGS - Flags for C++ compiler

Default:

CMAKE\_CXX\_FLAGS\_DEBUG - Flags used by the C++ compiler during debug builds

Default: -g

CMAKE\_CXX\_FLAGS\_MINSIZEREL - Flags used by the C++ compiler during release minsize builds

Default: -Os -DNDEBUG

CMAKE\_CXX\_FLAGS\_RELEASE - Flags used by the C++ compiler during release builds

Default: -O3 -DNDEBUG

CMAKE\_CXX\_STANDARD - The C++ standard to build C++ parts of SUNDIALS with.

Default: 11

Note: Options are 98, 11, 14, 17, 20. This option is on used when a C++ compiler is required.

CMAKE\_Fortran\_COMPILER - Fortran compiler

Default: /usr/bin/gfortran

Note: Fortran support (and all related options) are triggered only if either Fortran-C support is enabled (FCMIX\_ENABLE is ON) or LAPACK support is enabled (ENABLE\_LAPACK is ON).

CMAKE\_Fortran\_FLAGS - Flags for Fortran compiler

Default:

 ${\tt CMAKE\_Fortran\_FLAGS\_DEBUG\ -\ Flags\ used\ by\ the\ Fortran\ compiler\ during\ debug\ builds}$ 

Default: -g

 ${\tt CMAKE\_Fortran\_FLAGS\_MINSIZEREL~Flags~used~by~the~Fortran~compiler~during~release~minsize~builds}$ 

Default: -Os

 ${\tt CMAKE\_Fortran\_FLAGS\_RELEASE~-~Flags~used~by~the~Fortran~compiler~during~release~builds}$ 

Default: -O3

CMAKE\_INSTALL\_PREFIX - Install path prefix, prepended onto install directories

Default: /usr/local

Note: The user must have write access to the location specified through this option. Exported SUNDIALS header files and libraries will be installed under subdirectories include and CMAKE\_INSTALL\_LIBDIR of CMAKE\_INSTALL\_PREFIX, respectively.

CMAKE\_INSTALL\_LIBDIR - Library installation directory

Default:

Note: This is the directory within CMAKE\_INSTALL\_PREFIX that the SUNDIALS libraries will be installed under. The default is automatically set based on the operating system using the GNUInstallDirs CMake module.

Fortran\_INSTALL\_MODDIR - Fortran module installation directory

Default: fortran

ENABLE\_CUDA - Build the SUNDIALS CUDA modules.

Default: OFF

CUDA\_ARCH - Specifies the CUDA architecture to compile for.

Default:  $sm_30$ 

 ${\tt EXAMPLES\_ENABLE\_C}$  -  ${\tt Build}$  the  ${\tt SUNDIALS}$   ${\tt C}$  examples

Default: ON

 ${\tt EXAMPLES\_ENABLE\_CUDA}$  -  ${\tt Build}$  the  ${\tt SUNDIALS}$   ${\tt CUDA}$  examples

Default: OFF

Note: You need to enable CUDA support to build these examples.

 $EXAMPLES\_ENABLE\_CXX$  - Build the SUNDIALS  $C^{++}$  examples

Default: OFF unless ENABLE\_TRILINOS is ON.

EXAMPLES\_ENABLE\_F77 - Build the SUNDIALS Fortran77 examples

Default: ON (if F77\_INTERFACE\_ENABLE is ON)

EXAMPLES\_ENABLE\_F90 - Build the SUNDIALS Fortran90 examples

Default: ON (if F77\_INTERFACE\_ENABLE is ON)

EXAMPLES\_ENABLE\_F2003 - Build the SUNDIALS Fortran2003 examples

Default: ON (if BUILD\_FORTRAN\_MODULE\_INTERFACE is ON)

#### EXAMPLES\_INSTALL - Install example files

Default: ON

Note: This option is triggered when any of the SUNDIALS example programs are enabled (EXAMPLES\_ENABLE\_<language> is ON). If the user requires installation of example programs then the sources and sample output files for all SUNDIALS modules that are currently enabled will be exported to the directory specified by EXAMPLES\_INSTALL\_PATH. A CMake configuration script will also be automatically generated and exported to the same directory. Additionally, if the configuration is done under a Unix-like system, makefiles for the compilation of the example programs (using the installed SUNDIALS libraries) will be automatically generated and exported to the directory specified by EXAMPLES\_INSTALL\_PATH.

#### EXAMPLES\_INSTALL\_PATH - Output directory for installing example files

Default: /usr/local/examples

Note: The actual default value for this option will be an examples subdirectory created under CMAKE\_INSTALL\_PREFIX.

#### F77\_INTERFACE\_ENABLE - Enable Fortran-C support via the Fortran 77 interfaces

Default: OFF

#### BUILD\_FORTRAN\_MODULE\_INTERFACE - Enable Fortran-C support via the Fortran 2003 interfaces

Default: OFF

#### ENABLE\_HYPRE - Enable hypre support

Default: OFF

Note: See additional information on building with hypre enabled in A.1.4.

#### HYPRE\_INCLUDE\_DIR - Path to hypre header files

#### HYPRE\_LIBRARY\_DIR - Path to hypre installed library files

#### ENABLE\_KLU - Enable KLU support

Default: OFF

Note: See additional information on building with KLU enabled in A.1.4.

#### KLU\_INCLUDE\_DIR - Path to SuiteSparse header files

KLU\_LIBRARY\_DIR - Path to SuiteSparse installed library files

#### ENABLE\_LAPACK - Enable LAPACK support

Default: OFF

Note: Setting this option to ON will trigger additional CMake options. See additional information on building with LAPACK enabled in A.1.4.

#### LAPACK\_LIBRARIES - LAPACK (and BLAS) libraries

Default: /usr/lib/liblapack.so;/usr/lib/libblas.so

Note: CMake will search for libraries in your LD\_LIBRARY\_PATH prior to searching default system paths.

## ENABLE\_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.

#### ${\tt MPI\_C\_COMPILER\ -\ mpicc\ program}$

Default:

#### MPI\_CXX\_COMPILER - mpicxx program

Default:

Note: This option is triggered only if MPI is enabled (ENABLE\_MPI is ON) and C++ examples are enabled (EXAMPLES\_ENABLE\_CXX is ON). All SUNDIALS solvers can be used from C++ MPI applications by default without setting any additional configuration options other than ENABLE\_MPI.

MPI\_Fortran\_COMPILER - mpif77 or mpif90 program

Default:

Note: This option is triggered only if MPI is enabled (ENABLE\_MPI is ON) and Fortran-C support is enabled (F77\_INTERFACE\_ENABLE or BUILD\_FORTRAN\_MODULE\_INTERFACE is ON).

MPIEXEC\_EXECUTABLE - Specify the executable for running MPI programs

Default: mpirun

Note: This option is triggered only if MPI is enabled (ENABLE\_MPI is ON).

ENABLE\_OPENMP - Enable OpenMP support (build the OpenMP NVECTOR).

Default: OFF

OPENMP\_DEVICE\_ENABLE - Enable OpenMP device offloading (build the OpenMPDEV nvector) if supported by the provided compiler.

Default: OFF

OPENMP\_DEVICE\_WORKS - advanced option - Skip the check done to see if the OpenMP provided by the compiler supports OpenMP device offloading.

Default: OFF

ENABLE\_PETSC - Enable PETSc support

Default: OFF

Note: See additional information on building with PETSc enabled in ??.

PETSC\_DIR - Path to PETSc installation

Default:

PETSC\_LIBRARIES - advanced option - Semi-colon separated list of PETSc link libraries. Unless provided by the user, this is autopopulated based on the PETSc installation found in PETSC\_DIR. Default:

PETSC\_INCLUDES - advanced option - Semi-colon separated list of PETSc include directories. Unless provided by the user, this is autopopulated based on the PETSc installation found in PETSC\_DIR. Default:

ENABLE\_PTHREAD - Enable Pthreads support (build the Pthreads NVECTOR).

Default: OFF

ENABLE\_RAJA - Enable RAJA support.

Default: OFF

Note: You need to enable CUDA 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 and HIP.

Default: CUDA

ENABLE\_SUPERLUDIST - Enable SuperLU\_DIST support

Default: OFF

Note: See additional information on building with SuperLU\_DIST enabled in A.1.4.

SUPERLUDIST\_INCLUDE\_DIR - Path to SuperLU\_DIST header files (typically SRC directory)

SUPERLUDIST\_LIBRARY\_DIR - Path to SuperLU\_DIST installed library files

SUPERLUDIST\_LIBRARIES - Semi-colon separated list of libraries needed for SuperLU\_DIST

SUPERLUDIST\_OpenMP - Enable SUNDIALS support for SuperLU\_DIST built with OpenMP

Default: OFF

Note: SuperLU\_DIST must be built with OpenMP support for this option to function properly. Additionally the environment variable OMP\_NUM\_THREADS must be set to the desired number of threads.

ENABLE\_SUPERLUMT - Enable SUPERLUMT support

Default: OFF

Note: See additional information on building with Superlumt enabled in A.1.4.

SUPERLUMT\_INCLUDE\_DIR - Path to SuperLU\_MT header files (typically SRC directory)

SUPERLUMT\_LIBRARY\_DIR - Path to SuperLU\_MT installed library files

SUPERLUMT\_LIBRARIES - Semi-colon separated list of libraries needed for SuperLU\_MT

SUPERLUMT\_THREAD\_TYPE - Must be set to Pthread or OpenMP

Default: Pthread

ENABLE\_SYCL - Enable SYCL support.

Default: OFF

Note: At present the only supported SYCL compiler is the DPC++ (Intel oneAPI) compiler. CMake does not currently support autodetection of SYCL compilers and CMAKE\_CXX\_COMPILER must be set to a valid SYCL compiler i.e., dpcpp in order to build with SYCL support.

ENABLE\_TRILINOS - Enable Trilinos support (build the Tpetra NVECTOR).

Default: OFF

Trilinos\_DIR - Path to the Trilinos install directory.

Default:

TRILINOS\_INTERFACE\_C\_COMPILER - advanced option - Set the C compiler for building the Trilinos interface (i.e., NVECTOR\_TRILINOS and the examples that use it).

Default: The C compiler exported from the found Trilinos installation if USE\_XSDK\_DEFAULTS=OFF.

CMAKE\_C\_COMPILER or MPI\_C\_COMPILER if USE\_XSDK\_DEFAULTS=ON.

Note: It is recommended to use the same compiler that was used to build the Trilinos library.

TRILINOS\_INTERFACE\_C\_COMPILER\_FLAGS - advanced option - Set the C compiler flags for Trilinos interface (i.e., NVECTOR\_TRILINOS and the examples that use it).

Default: The C compiler flags exported from the found Trilinos installation if USE\_XSDK\_DEFAULTS=OFF. CMAKE\_C\_FLAGS if USE\_XSDK\_DEFAULTS=ON.

Note: It is recommended to use the same flags that were used to build the Trilinos library.

TRILINOS\_INTERFACE\_CXX\_COMPILER - advanced option - Set the C++ compiler for builing Trilinos interface (i.e., NVECTOR\_TRILINOS and the examples that use it).

Default: The  $C^{++}$  compiler exported from the found Trilinos installation if USE\_XSDK\_DEFAULTS=OFF. CMAKE\_CXX\_COMPILER or MPI\_CXX\_COMPILER if USE\_XSDK\_DEFAULTS=ON.

Note: It is recommended to use the same compiler that was used to build the Trilinos library.

TRILINOS\_INTERFACE\_CXX\_COMPILER\_FLAGS - advanced option - Set the C++ compiler flags for Trilinos interface (i.e., NVECTOR\_TRILINOS and the examples that use it).

Default: The  $C^{++}$  compiler flags exported from the found Trilinos installation if USE\_XSDK\_DEFAULTS=OFF. CMAKE\_CXX\_FLAGS if USE\_XSDK\_DEFAULTS=ON.

Note: Is is recommended to use the same flags that were used to build the Trilinos library.

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\_PACKAGE\_FUSED\_KERNELS - Build specialized fused kernels inside CVODE.

Default: OFF

Note: This option is currently only available when building with CUDA\_ENABLE = ON. Building with fused kernels requires linking to either libsundials\_cvode\_fused\_cuda. lib or libsundials\_cvode\_fused\_stude where the latter provides CPU-only placeholders for the fused routines, in addition to libsundials\_cvode. lib.

CMAKE\_CXX\_STANDARD - The C++ standard to build C++ parts of SUNDIALS with.

Default: 11

Note: Options are 99, 11, 14, 17. This option only used when a C++ compiler is required.

SUNDIALS\_F77\_FUNC\_CASE - advanced option - Specify the case to use in the Fortran name-mangling scheme, options are: lower or upper

Default:

Note: The build system will attempt to infer the Fortran name-mangling scheme using the Fortran compiler. This option should only be used if a Fortran compiler is not available or to override the inferred or default (lower) scheme if one can not be determined. If used, SUNDIALS\_F77\_FUNC\_UNDERSCORES must also be set.

SUNDIALS\_F77\_FUNC\_UNDERSCORES - advanced option - Specify the number of underscores to append in the Fortran name-mangling scheme, options are: none, one, or two Default:

Note: The build system will attempt to infer the Fortran name-mangling scheme using the Fortran compiler. This option should only be used if a Fortran compiler is not available or to override the inferred or default (one) scheme if one can not be determined. If used, SUNDIALS\_F77\_FUNC\_CASE must also be set.

 ${\tt SUNDIALS\_INDEX\_TYPE - advanced\ option - Integer\ type\ used\ for\ SUNDIALS\ indices.} \ The\ size\ must match\ the\ size\ provided\ for\ the}$ 

SUNDIALS\_INDEX\_SIZE option.

Default

Note: In past SUNDIALS versions, a user could set this option to INT64\_T to use 64-bit integers, or INT32\_T to use 32-bit integers. Starting in SUNDIALS 3.2.0, these special values are deprecated. For SUNDIALS 3.2.0 and up, a user will only need to use the SUNDIALS\_INDEX\_SIZE option in most cases.

SUNDIALS\_INDEX\_SIZE - Integer size (in bits) used for indices in SUNDIALS, options are: 32 or 64 Default: 64

Note: The build system tries to find an integer type of appropriate size. Candidate 64-bit integer types are (in order of preference): int64\_t, \_\_int64, long long, and long. Candidate 32-bit integers are (in order of preference): int32\_t, int, and long. The advanced option, SUNDIALS\_INDEX\_TYPE can be used to provide a type not listed here.

SUNDIALS\_PRECISION - Precision used in SUNDIALS, options are: double, single, or extended Default: double

SUNDIALS\_INSTALL\_CMAKEDIR - Installation directory for the SUNDIALS cmake files (relative to CMAKE\_INSTALL\_PREFIX).

Default: CMAKE\_INSTALL\_PREFIX/cmake/sundials

 $\begin{array}{ll} {\tt USE\_GENERIC\_MATH\ -\ Use\ generic\ (stdc)\ math\ libraries} \\ {\tt Default:\ ON} \end{array}$ 

USE\_XSDK\_DEFAULTS - Enable xSDK (see for more information) default configuration settings. This sets CMAKE\_BUILD\_TYPE to Debug, SUNDIALS\_INDEX\_SIZE to 32 and SUNDIALS\_PRECISION to double.

Default: OFF

#### A.1.3 Configuration examples

The following examples will help demonstrate usage of the CMake configure options.

To configure SUNDIALS using the default C and Fortran compilers, and default mpic and mpif77 parallel compilers, enable compilation of examples, and install libraries, headers, and example sources under subdirectories of /home/myname/sundials/, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DENABLE_MPI=ON \
> -DFCMIX_ENABLE=ON \
> /home/myname/sundials/solverdir
%
% make install
%
```

To disable installation of the examples, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DENABLE_MPI=ON \
> -DFCMIX_ENABLE=ON \
> -DEXAMPLES_INSTALL=OFF \
> /home/myname/sundials/solverdir
%
% make install
%
```

#### A.1.4 Working with external Libraries

The SUNDIALS suite contains many options to enable implementation flexibility when developing solutions. The following are some notes addressing specific configurations when using the supported third party libraries. When building SUNDIALS as a shared library any external libraries used with SUNDIALS must also be build as a shared library or as a static library compiled with the -fPIC flag.



#### Building with LAPACK

To enable LAPACK, set the ENABLE\_LAPACK option to ON. If the directory containing the LAPACK library is in the LD\_LIBRARY\_PATH environment variable, CMake will set the LAPACK\_LIBRARIES variable accordingly, otherwise CMake will attempt to find the LAPACK library in standard system locations. To explicitly tell CMake what library to use, the LAPACK\_LIBRARIES variable can be set to the desired libraries rquired for LAPACK.

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DENABLE_LAPACK=ON \
> -DLAPACK_LIBRARIES=/mylapackpath/lib/libblas.so;/mylapackpath/lib/liblapack.so \
> /home/myname/sundials/solverdir
%
make install
%
```

If a working Fortran compiler is not available to infer the Fortran name-mangling scheme, the options SUNDIALS\_F77\_FUNC\_CASE and SUNDIALS\_F77\_FUNC\_UNDERSCORES must be set in order to bypass the check for a Fortran compiler and define the name-mangling scheme. The defaults for these options in earlier versions of SUNDIALS were lower and one respectively.

#### Building with KLU

The KLU libraries are part of SuiteSparse, a suite of sparse matrix software, available from the Texas A&M University website: http://faculty.cse.tamu.edu/davis/suitesparse.html. SUNDIALS has been tested with SuiteSparse version 5.7.2. To enable KLU, set ENABLE\_KLU to ON, set KLU\_INCLUDE\_DIR to the include path of the KLU installation and set KLU\_LIBRARY\_DIR to the lib path of the KLU installation. The CMake configure will result in populating the following variables: AMD\_LIBRARY, AMD\_LIBRARY\_DIR, BTF\_LIBRARY\_DIR, COLAMD\_LIBRARY, COLAMD\_LIBRARY\_DIR, and KLU\_LIBRARY.

#### Building with SuperLU\_MT

The SuperLU\_MT libraries are available for download from the Lawrence Berkeley National Laboratory website: http://crd-legacy.lbl.gov/~xiaoye/SuperLU/#superlu\_mt. SUNDIALS has been tested with SuperLU\_MT version 3.1. To enable SuperLU\_MT, set ENABLE\_SUPERLUMT to ON, set SUPERLUMT\_INCLUDE\_DIR to the SRC path of the SuperLU\_MT installation, and set the variable SUPERLUMT\_LIBRARY\_DIR to the lib path of the SuperLU\_MT installation. At the same time, the variable SUPERLUMT\_LIBRARIES must be set to a semi-colon separated list of other libraries SuperLU\_MT depends on. For example, if SuperLU\_MT ws build with an external blas library, then include the full path to the blas library in this list. Additionally, the variable SUPERLUMT\_THREAD\_TYPE must be set to either Pthread or OpenMP.



Do not mix thread types when building SUNDIALS solvers. If threading is enabled for SUNDIALS by having either ENABLE\_OPENMP or ENABLE\_PTHREAD set to ON then SuperLU\_MT should be set to use the same threading type.

#### Building with SuperLU\_DIST

The SuperLU\_DIST libraries are available for download from the Lawrence Berkeley National Laboratory website: http://crd-legacy.lbl.gov/~xiaoye/SuperLU/#superlu\_dist. SUNDIALS has been tested with SuperLU\_DIST 6.1.1. To enable SuperLU\_DIST, set ENABLE\_SUPERLUDIST to ON, set SUPERLUDIST\_INCLUDE\_DIR to the include directory of the SuperLU\_DIST installation (typically SRC), and set the variable

SUPERLUDIST\_LIBRARY\_DIR to the path to library directory of the SuperLU\_DIST installation (typically lib). At the same time, the variable SUPERLUDIST\_LIBRARIES must be set to a semi-colon separated list of other libraries SuperLU\_DIST depends on. For example, if SuperLU\_DIST was built with LAPACK, then include the LAPACK library in this list. If SuperLU\_DIST was built with OpenMP support, then you may set SUPERLUDIST\_OPENMP to ON to utilize the OpenMP functionality of SuperLU\_DIST.



Do not mix thread types when building SUNDIALS solvers. If threading is enabled for SUNDIALS by having ENABLE\_PTHREAD set to ON then SuperLU\_DIST should not be set to use OpenMP.

#### Building with PETSc

The PETSc libraries are available for download from the Argonne National Laboratory website: http://www.mcs.anl.gov/SUNDIALS has been tested with PETSc version 3.10.0-3.14.0. To enable PETSc, set ENABLE\_PETSC to ON and then set PETSC\_DIR to the path of the PETSc installation. Alternatively, a user can provide a list of include paths in PETSC\_INCLUDES, and a list of complete paths to the libraries needed in PETSC\_LIBRARIES.

#### Building with hypre

The hypre libraries are available for download from the Lawrence Livermore National Laboratory website: http://computing.llnl.gov/projects/hypre. SUNDIALS has been tested with hypre version 2.14.0-2.19.0. To enable hypre, set ENABLE\_HYPRE to ON, set HYPRE\_INCLUDE\_DIR to the include path of the hypre installation, and set the variable HYPRE\_LIBRARY\_DIR to the lib path of the hypre installation.

Note: SUNDIALS must be configured so that SUNDIALS\_INDEX\_SIZE (or equivalently, XSDK\_INDEX\_SIZE) equals the precision of HYPRE\_BigInt in the corresponding hypre installation.

#### Building with CUDA

SUNDIALS CUDA modules and examples have been tested with versions 9 through 11.0.2 of the CUDA toolkit. To build them, you need to install the Toolkit and compatible NVIDIA drivers. Both are available for download from the NVIDIA website: https://developer.nvidia.com/cuda-downloads. To enable CUDA, set ENABLE\_CUDA to ON. If CUDA is installed in a nonstandard location, you may be prompted to set the variable CUDA\_TOOLKIT\_ROOT\_DIR with your CUDA Toolkit installation path. To enable CUDA examples, set EXAMPLES\_ENABLE\_CUDA to ON.

#### Building with RAJA

RAJA is a performance portability layer developed by Lawrence Livermore National Laboratory and can be obtained from https://github.com/LLNL/RAJA. SUNDIALS RAJA modules and examples have been tested with RAJA up to version 0.12.1. Building SUNDIALS RAJA modules requires a CUDA-enabled RAJA installation. To enable RAJA, set ENABLE\_CUDA and ENABLE\_RAJA to ON. If RAJA is installed in a nonstandard location you will be prompted to set the variable RAJA\_DIR with the path to the RAJA CMake configuration file. To enable building the RAJA examples set EXAMPLES\_ENABLE\_CUDA to ON.

#### **Building with Trilinos**

Trilinos is a suite of numerical libraries developed by Sandia National Laboratories. It can be obtained at https://github.com/trilinos/Trilinos. SUNDIALS Trilinos modules and examples have been tested with Trilinos version 12.14.1 – 12.18.1. To enable Trilinos, set ENABLE\_TRILINOS to ON. If Trilinos is installed in a nonstandard location you will be prompted to set the variable Trilinos\_DIR with the path to the Trilinos CMake configuration file. It is desireable to build the Trilinos vector interface with same compiler and options that were used to build Trilinos. CMake will try to find the correct compiler settings automatically from the Trilinos configuration file. If that is not successful, the compilers and options can be manually set with the following CMake variables:

- Trilinos\_INTERFACE\_C\_COMPILER
- Trilinos\_INTERFACE\_C\_COMPILER\_FLAGS
- Trilinos\_INTERFACE\_CXX\_COMPILER
- Trilinos\_INTERFACE\_CXX\_COMPILER\_FLAGS

#### A.1.5 Testing the build and installation

If SUNDIALS was configured with EXAMPLES\_ENABLE\_<language> options to ON, then a set of regression tests can be run after building with the make command by running:

```
% make test
```

Additionally, if EXAMPLES\_INSTALL was also set to ON, then a set of smoke tests can be run after installing with the make install command by running:

```
% make test_install
```

### A.2 Building and Running Examples

Each of the SUNDIALS solvers is distributed with a set of examples demonstrating basic usage. To build and install the examples, set at least of the EXAMPLES\_ENABLE\_<language> options to ON, and set EXAMPLES\_INSTALL to ON. Specify the installation path for the examples with the variable EXAMPLES\_INSTALL\_PATH. CMake will generate CMakeLists.txt configuration files (and Makefile files if on Linux/Unix) that reference the *installed* SUNDIALS headers and libraries.

Either the CMakeLists.txt file or the traditional Makefile may be used to build the examples as well as serve as a template for creating user developed solutions. To use the supplied Makefile simply run make to compile and generate the executables. To use CMake from within the installed example directory, run cmake (or ccmake to use the GUI) followed by make to compile the example code. Note that if CMake is used, it will overwrite the traditional Makefile with a new CMake-generated Makefile. The resulting output from running the examples can be compared with example output bundled in the SUNDIALS distribution.



NOTE: There will potentially be differences in the output due to machine architecture, compiler versions, use of third party libraries etc.

### A.3 Configuring, building, and installing on Windows

CMake can also be used to build SUNDIALS on Windows. To build SUNDIALS for use with Visual Studio the following steps should be performed:

- 1. Unzip the downloaded tar file(s) into a directory. This will be the solverdir
- 2. Create a separate builddir
- 3. Open a Visual Studio Command Prompt and cd to builddir
- 4. Run cmake-gui ../solverdir
  - (a) Hit Configure
  - (b) Check/Uncheck solvers to be built
  - (c) Change CMAKE\_INSTALL\_PREFIX to instdir
  - (d) Set other options as desired
  - (e) Hit Generate
- 5. Back in the VS Command Window:
  - (a) Run msbuild ALL\_BUILD.vcxproj
  - (b) Run msbuild INSTALL.vcxproj

The resulting libraries will be in the *instdir*. The SUNDIALS project can also now be opened in Visual Studio. Double click on the ALL\_BUILD.vcxproj file to open the project. Build the whole *solution* to create the SUNDIALS libraries. To use the SUNDIALS libraries in your own projects, you must set the include directories for your project, add the SUNDIALS libraries to your project solution, and set the SUNDIALS libraries as dependencies for your project.

## A.4 Installed libraries and exported header files

Using the CMake SUNDIALS build system, the command

will install the libraries under *libdir* and the public header files under *includedir*. The values for these directories are *instdir*/CMAKE\_INSTALL\_LIBDIR and *instdir*/include, respectively. The location can be changed by setting the CMake variable CMAKE\_INSTALL\_PREFIX. Although all installed libraries reside under *libdir*/CMAKE\_INSTALL\_LIBDIR, the public header files are further organized into subdirectories under *includedir*/include.

The installed libraries and exported header files are listed for reference in Table A.1. The file extension .lib is typically .so for shared libraries and .a for static libraries. Note that, in the Tables, names are relative to 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 <code>includedir/include/sundials</code> directory since they are explicitly included by the appropriate solver header files (e.g., <code>cvode\_dense.h</code> includes <code>sundials\_dense.h</code>). However, it is both legal and safe to do so, and would be useful, for example, if the functions declared in <code>sundials\_dense.h</code> are to be used in building a preconditioner.

#### A.4.1 Using Sundials as a Third Party Library in other CMake Projects

The make install command will also install a CMake package configuration file that other CMake projects can load to get all the information needed to build against SUNDIALS. In the consuming project's CMake code, the find\_package command may be used to search for the configuration file, which will be installed to instdir/SUNDIALS\_INSTALL\_CMAKEDIR/SUNDIALSConfig.cmake alongside a package version file instdir/SUNDIALS\_INSTALL\_CMAKEDIR/SUNDIALSConfigVersion.cmake. Together these files contain all the information the consuming project needs to use SUNDIALS, including exported CMake targets. The SUNDIALS exported CMake targets follow the same naming convention as the generated library binaries, e.g. the exported target for CVODE is SUNDIALS::cvode. The CMake code snipped below shows how a consuming project might leverage the SUNDIALS package configuration file to build against SUNDIALS in their own CMake project.

```
# 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_project(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)
```

Tak		ALS libraries and header files
SHARED	Libraries	n/a
	Header files	sundials/sundials_config.h
		sundials/sundials_fconfig.h
		sundials/sundials_types.h
		sundials/sundials_math.h
		sundials/sundials_nvector.h
		sundials/sundials_fnvector.h
		sundials/sundials_matrix.h
		sundials/sundials_linearsolver.h
		sundials/sundials_iterative.h
		sundials/sundials_direct.h
		sundials/sundials_dense.h
		sundials/sundials_band.h
		sundials/sundials_nonlinearsolver.h
		sundials/sundials_version.h
		sundials/sundials_mpi_types.h
		sundials/sundials_cuda_policies.hpp
NVECTOR_SERIAL	Libraries	libsundials_nvecserial.lib
		libsundials_fnvecserial_mod.lib
		libsundials_fnvecserial.a
	Header files	nvector/nvector_serial.h
	Module	fnvector_serial_mod.mod
	files	
NVECTOR_PARALLEL	Libraries	libsundials_nvecparallel.lib
		libsundials_fnvecparallel.a
		libsundials_fnvecparallel_mod.lib
	Header files	nvector/nvector_parallel.h
	Module	fnvector_parallel_mod.mod
	files	
NVECTOR_MANYVECTOR	Libraries	libsundials_nvecmanyvector.lib
		libsundials_nvecmanyvector_mod.lib
	Header files	nvector/nvector_manyvector.h
	Module	fnvector_manyvector_mod.mod
	files	
NVECTOR_MPIMANYVECTOR	Libraries	libsundials_nvecmpimanyvector.lib
		libsundials_nvecmpimanyvector_mod.lib
	Header files	nvector/nvector_mpimanyvector.h
	Module	fnvector_mpimanyvector_mod.mod
	files	•
	•	continued on next page

continued from last page	T *1 *	1:1 1:1 :1 1:7
NVECTOR_MPIPLUSX	Libraries	libsundials_nvecmpiplusx.lib
	TT 1 C1	libsundials_nvecmpiplusx_mod.lib
	Header files	nvector/nvector_mpiplusx.h
	Module	fnvector_mpiplusx_mod.mod
	files	
NVECTOR_OPENMP	Libraries	libsundials_nvecopenmp.lib
		$libsundials\_fnvecopenmp\_mod.lib$
		libsundials_fnvecopenmp.a
	Header files	nvector/nvector_openmp.h
	Module	fnvector_openmp_mod.mod
	files	
NVECTOR_OPENMPDEV	Libraries	$libsundials\_nvecopenmpdev. \it lib$
	Header files	nvector/nvector_openmpdev.h
NVECTOR_PTHREADS	Libraries	$libsundials\_nvecpthreads.lib$
		$libsundials\_fnvecpthreads\_mod. \it lib$
		libsundials_fnvecpthreads.a
	Header files	nvector/nvector_pthreads.h
	Module	fnvector_pthreads_mod.mod
	files	
NVECTOR_PARHYP	Libraries	libsundials_nvecparhyp.lib
	Header files	nvector/nvector_parhyp.h
NVECTOR_PETSC	Libraries	libsundials_nvecpetsc.lib
	Header files	nvector/nvector_petsc.h
NVECTOR_CUDA	Libraries	libsundials_nveccuda.lib
	Header files	nvector/nvector_cuda.h
NVECTOR_HIP	Libraries	libsundials_nvechip.lib
	Header files	nvector/nvector_hip.h
NVECTOR_RAJA	Libraries	libsundials_nveccudaraja.lib
		libsundials_nvechipraja.lib
	Header files	nvector/nvector_raja.h
NVECTOR_SYCL	Libraries	libsundials_nvecsycl.lib
	Header files	nvector/nvector_sycl.h
NVECTOR_TRILINOS	Libraries	libsundials_nvectrilinos.lib
	Header files	nvector/nvector_trilinos.h
		nvector/trilinos/SundialsTpetraVectorInterface.hpp
		nvector/trilinos/SundialsTpetraVectorKernels.hpp
SUNMATRIX_BAND	Libraries	libsundials_sunmatrixband.lib
		libsundials_fsunmatrixband_mod.lib
		libsundials_fsunmatrixband.a
	Header files	sunmatrix/sunmatrix_band.h
	Module	fsunmatrix_band_mod.mod
	files	Dallimovi III-Jourd III Od III Od
		continued on next p

continued from last page		
SUNMATRIX_DENSE	Libraries	libsundials_sunmatrixdense.lib
		libsundials_fsunmatrixdense_mod.lib
		libsundials_fsunmatrixdense.a
	Header files	sunmatrix/sunmatrix_dense.h
	Module	fsunmatrix_dense_mod.mod
	files	
SUNMATRIX_SPARSE	Libraries	libsundials_sunmatrixsparse.lib
		$lib sundials\_f sunmatrix sparse\_mod. lib$
		libsundials_fsunmatrixsparse.a
	Header files	sunmatrix/sunmatrix_sparse.h
	Module	fsunmatrix_sparse_mod.mod
	files	
SUNMATRIX_SLUNRLOC	Libraries	libsundials_sunmatrixslunrloc.lib
	Header files	sunmatrix/sunmatrix_slunrloc.h
SUNLINSOL_CUSPARSE	Libraries	libsundials_sunmatrixcusparse.lib
	Header files	sunmatrix/sunmatrix_cusparse.h
SUNLINSOL_BAND	Libraries	libsundials_sunlinsolband.lib
		libsundials_fsunlinsolband_mod.lib
		libsundials_fsunlinsolband.a
	Header files	sunlinsol/sunlinsol_band.h
	Module	fsunlinsol_band_mod.mod
	files	
SUNLINSOL_DENSE	Libraries	libsundials_sunlinsoldense.lib
		$libsundials\_fsunlinsoldense\_mod. lib$
		libsundials_fsunlinsoldense.a
	Header files	sunlinsol/sunlinsol_dense.h
	Module	fsunlinsol_dense_mod.mod
	files	
SUNLINSOL_KLU	Libraries	libsundials_sunlinsolklu.lib
		$libsundials\_fsunlinsolklu\_mod. \it lib$
		libsundials_fsunlinsolklu.a
	Header files	sunlinsol/sunlinsol_klu.h
	Module	fsunlinsol_klu_mod.mod
	files	
SUNLINSOL_LAPACKBAND	Libraries	$libsundials\_sunlinsollapackband. lib$
		libsundials_fsunlinsollapackband.a
	Header files	sunlinsol/sunlinsol_lapackband.h
SUNLINSOL_LAPACKDENSE	Libraries	$libsundials\_sunlinsollapackdense. \it lib$
		libsundials_fsunlinsollapackdense.a
	Header files	sunlinsol/sunlinsol_lapackdense.h
SUNLINSOL_PCG	Libraries	libsundials_sunlinsolpcg.lib
		$libsundials\_fsunlinsolpcg\_mod. lib$
		continued on next page

continued from last page		
continued from fact page		libsundials_fsunlinsolpcg.a
	Header files	sunlinsol/sunlinsol_pcg.h
	Module	fsunlinsol_pcg_mod.mod
	files	is difficult of the control of the c
SUNLINSOL_SPBCGS	Libraries	libsundials_sunlinsolspbcgs.lib
		libsundials_fsunlinsolspbcgs_mod.lib
		libsundials_fsunlinsolspbcgs.a
	Header files	sunlinsol/sunlinsol_spbcgs.h
	Module	fsunlinsol_spbcgs_mod.mod
	files	
SUNLINSOL_SPFGMR	Libraries	libsundials_sunlinsolspfgmr.lib
		$libsundials\_fsunlinsolspfgmr\_mod. \it lib$
		libsundials_fsunlinsolspfgmr.a
	Header files	sunlinsol/sunlinsol_spfgmr.h
	Module	fsunlinsol_spfgmr_mod.mod
	files	
SUNLINSOL_SPGMR	Libraries	libsundials_sunlinsolspgmr.lib
		libsundials_fsunlinsolspgmr_mod.lib
		libsundials_fsunlinsolspgmr.a
	Header files	sunlinsol/sunlinsol_spgmr.h
	Module	fsunlinsol_spgmr_mod.mod
	files	
SUNLINSOL_SPTFQMR	Libraries	libsundials_sunlinsolsptfqmr.lib
		$libsundials\_fsunlinsolsptfqmr\_mod. lib$
		libsundials_fsunlinsolsptfqmr.a
	Header files	sunlinsol/sunlinsol_sptfqmr.h
	Module	fsunlinsol_sptfqmr_mod.mod
	files	
SUNLINSOL_SUPERLUMT	Libraries	$libsundials\_sunlinsolsuperlumt.lib$
		libsundials_fsunlinsolsuperlumt.a
	Header files	sunlinsol/sunlinsol_superlumt.h
SUNLINSOL_SUPERLUDIST	Libraries	$libsundials\_sunlinsolsuperludist. \it lib$
	Header files	sunlinsol/sunlinsol_superludist.h
SUNLINSOL_CUSOLVERSP_BAT	-	$libsundials\_sunlinsolcusolversp. \it lib$
	Header files	sunlinsol/sunlinsol_cusolverp_batchqr.h
SUNNONLINSOL_NEWTON	Libraries	$lib sundials\_sunnon linsol newton. \it lib$
		$libsundials\_fsunnonlinsolnewton\_mod. lib$
		libsundials_fsunnonlinsolnewton.a
	Header files	sunnonlinsol/sunnonlinsol_newton.h
	Module	fsunnonlinsol_newton_mod.mod
	files	
SUNNONLINSOL_FIXEDPOINT	Libraries	$lib sundials\_sunnon lins of fixed point. \it lib$
		continued on next page

continued from last page			
		libsundials_fsunnonlinsolfixe	edpoint.a
		libsundials_fsunnonlinsolfixe	
	Header files	sunnonlinsol/sunnonlinsol_fi	
	Module	fsunnonlinsol_fixedpoint_mo	
	files		
SUNNONLINSOL_PETSCSNES	Libraries	libsundials_sunnonlinsolpets	csnes. $lib$
	Header files	sunnonlinsol/sunnonlinsol_petscsnes.h	
CVODE	Libraries	libsundials_cvode.lib	
		libsundials_fcvode.a	
		$libsundials\_fcvode\_mod.lib$	
	Header files	cvode/cvode.h	cvode/cvode_impl.h
		cvode/cvode_direct.h	cvode/cvode_ls.h
		cvode/cvode_spils.h	cvode/cvode_bandpre.h
		cvode/cvode_bbdpre.h	,
	Module	fcvode_mod.mod	
	files		
CVODES	Libraries	libsundials_cvodes.lib	
		libsundials_fcvodes_mod.lib	
	Header files	cvodes/cvodes.h	cvodes/cvodes_impl.h
		cvodes/cvodes_direct.h	cvodes/cvodes_ls.h
		cvodes/cvodes_spils.h	cvodes/cvodes_bandpre.h
		cvodes/cvodes_bbdpre.h	,
	Module	fcvodes_mod.mod	
	files		
ARKODE	Libraries	$libsundials\_arkode.lib$	
		libsundials_farkode.a	
		libsundials_farkode_mod.lib	
	Header files	arkode/arkode.h	arkode/arkode_impl.h
		arkode/arkode_ls.h	arkode/arkode_bandpre.h
		arkode/arkode_bbdpre.h	
	Module	farkode_mod.mod	farkode_arkstep_mod.mod
	files		
		farkode_erkstep_mod.mod	$farkode\_mristep\_mod.mod$
IDA	Libraries	libsundials_ida.lib	
		libsundials_fida.a	
		$libsundials\_fida\_mod.lib$	
	Header files	ida/ida.h	ida/ida_impl.h
		ida/ida_direct.h	$ida/ida_ls.h$
		ida/ida_spils.h	ida/ida_bbdpre.h
	Module	fida_mod.mod	
	files		
			continued on next page

continued from last page			
IDAS	Libraries	libsundials_idas.lib	
		$libsundials\_fidas\_mod.lib$	
	Header files	idas/idas.h	idas/idas_impl.h
		idas/idas_direct.h	$idas/idas_ls.h$
		idas/idas_spils.h	$idas/idas\_bbdpre.h$
	Module	fidas_mod.mod	
	files		
KINSOL	Libraries	libsundials_kinsol.lib	
		libsundials_fkinsol.a	
		libsundials_fkinsol_mod. $lib$	
	Header files	kinsol/kinsol.h	kinsol/kinsol_impl.h
		kinsol/kinsol_direct.h	kinsol/kinsol_ls.h
		kinsol/kinsol_spils.h	$kinsol/kinsol\_bbdpre.h$
	Module	fkinsol_mod.mod	
	files		

## Appendix B

# **IDA** Constants

Below we list all input and output constants used by the main solver and linear solver modules, together with their numerical values and a short description of their meaning.

## B.1 IDA input constants

IDA main solver module			
IDA_NORMAL	1	Solver returns at specified output time.	
IDA_ONE_STEP	2	Solver returns after each successful step.	
IDA_YA_YDP_INIT	1	Compute $y_a$ and $\dot{y}_d$ , given $y_d$ .	
IDA_Y_INIT	2	Compute $y$ , given $\dot{y}$ .	
	Ite	rative linear solver module	
PREC_NONE	0	No preconditioning	
PREC_LEFT	1	Preconditioning on the left.	
${ t MODIFIED\_GS}$	1	Use modified Gram-Schmidt procedure.	
CLASSICAL_GS	2	Use classical Gram-Schmidt procedure.	

## B.2 IDA output constants

IDA 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.	
TDA TOO MIGII AGG	0	*****	
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.	

378 IDA Constants

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 re- coverable 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 IDA 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.

#### 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 NVECTOR
		module.
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 Sunlinsol module.

# Appendix C

# SUNDIALS Release History

Table C.1: Release History

Date		SUNDIALS	ARKODE	CVODE	CVODES	IDA	IDAS	KINSOL
	2021		170	<b>.</b>			4.50	
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
continued on next page								

continued from last page								
Da	ate	SUNDIALS	ARKODE	CVODE	CVODES	IDA	IDAS	KINSOL
Mar	2012	2.5.0	_	2.7.0	2.7.0	2.7.0	1.1.0	2.7.0
May	2009	2.4.0	_	2.6.0	2.6.0	2.6.0	1.0.0	2.6.0
Nov	2006	2.3.0	_	2.5.0	2.5.0	2.5.0	_	2.5.0
Mar	2006	2.2.0	_	2.4.0	2.4.0	2.4.0	_	2.4.0
May	2005	2.1.1	_	2.3.0	2.3.0	2.3.0	_	2.3.0
Apr	2005	2.1.0	_	2.3.0	2.2.0	2.3.0	_	2.3.0
Mar	2005	2.0.2	_	2.2.2	2.1.2	2.2.2	_	2.2.2
Jan	2005	2.0.1	_	2.2.1	2.1.1	2.2.1	_	2.2.1
Dec	2004	2.0.0	_	2.2.0	2.1.0	2.2.0	_	2.2.0
Jul	2002	1.0.0	_	2.0.0	1.0.0	2.0.0	_	2.0.0
Mar	2002	_	_	$1.0.0^{3}$	_	_	_	_
Feb	1999	_	_	_	_	$1.0.0^4$	_	_
Aug	1998	_	_	_	_	_	_	$1.0.0^{5}$
Jul	1997	_	_	$1.0.0^2$	_	_	_	_
Sep	1994	_	_	$1.0.0^{1}$	_	_	_	_
<sup>1</sup> CVODE written, <sup>2</sup> PVODE written, <sup>3</sup> CVODE and PVODE combined, <sup>4</sup> IDA written, <sup>5</sup> KINSOL written								

## Bibliography

- [1] AMD ROCm Documentation. https://rocmdocs.amd.com/en/latest/index.html.
- [2] Intel one API Programming Guide. https://software.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html.
- [3] KLU Sparse Matrix Factorization Library. http://faculty.cse.tamu.edu/davis/suitesparse.html.
- [4] MAGMA: Matrix Algebra on GPU and Multicore Architectures. https://icl.utk.edu/magma/index.html.
- [5] NVIDIA CUDA Programming Guide. https://docs.nvidia.com/cuda/index.html.
- [6] NVIDIA cuSOLVER Programming Guide. https://docs.nvidia.com/cuda/cusolver/index.html.
- [7] NVIDIA cuSPARSE Programming Guide. https://docs.nvidia.com/cuda/cusparse/index.html.
- [8] SuperLU\_DIST Parallel Sparse Matrix Factorization Library. http://crd-legacy.lbl.gov/xiaoye/-SuperLU/.
- [9] SuperLU\_MT Threaded Sparse Matrix Factorization Library. http://crd-legacy.lbl.gov/xiaoye/-SuperLU/.
- [10] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.12, Argonne National Laboratory, 2019.
- [11] K. E. Brenan, S. L. Campbell, and L. R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. SIAM, Philadelphia, Pa, 1996.
- [12] P. N. Brown. A local convergence theory for combined inexact-Newton/finite difference projection methods. SIAM J. Numer. Anal., 24(2):407–434, 1987.
- [13] P. N. Brown and A. C. Hindmarsh. Reduced Storage Matrix Methods in Stiff ODE Systems. *J. Appl. Math. & Comp.*, 31:49–91, 1989.
- [14] 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.
- [15] 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.
- [16] P. N. Brown and Y. Saad. Hybrid Krylov Methods for Nonlinear Systems of Equations. SIAM J. Sci. Stat. Comput., 11:450–481, 1990.
- [17] G. D. Byrne. Pragmatic Experiments with Krylov Methods in the Stiff ODE Setting. In J.R. Cash and I. Gladwell, editors, Computational Ordinary Differential Equations, pages 323–356, Oxford, 1992. Oxford University Press.

382 BIBLIOGRAPHY

[18] 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.

- [19] 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.
- [20] S. D. Cohen and A. C. Hindmarsh. CVODE, a Stiff/Nonstiff ODE Solver in C. Computers in Physics, 10(2):138–143, 1996.
- [21] A. M. Collier, A. C. Hindmarsh, R. Serban, and C.S. Woodward. User Documentation for KINSOL v5.7.0. Technical Report UCRL-SM-208116, LLNL, 2021.
- [22] T. A. Davis and P. N. Ekanathan. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Softw.*, 37(3), 2010.
- [23] R. S. Dembo, S. C. Eisenstat, and T. Steihaug. Inexact Newton Methods. SIAM J. Numer. Anal., 19:400–408, 1982.
- [24] 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.
- [25] J. E. Dennis and R. B. Schnabel. Numerical Methods for Unconstrained Optimization and Nonlinear Equations. SIAM, Philadelphia, 1996.
- [26] 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.
- [27] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. SIAM J. Sci. Comp., 14:470–482, 1993.
- [28] 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.
- [29] 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.
- [30] K. L. Hiebert and L. F. Shampine. Implicitly Defined Output Points for Solutions of ODEs. Technical Report SAND80-0180, Sandia National Laboratories, February 1980.
- [31] A. C. Hindmarsh. The PVODE and IDA Algorithms. Technical Report UCRL-ID-141558, LLNL, December 2000.
- [32] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS, suite of nonlinear and differential/algebraic equation solvers. ACM Trans. Math. Softw., (31):363–396, 2005.
- [33] A. C. Hindmarsh and R. Serban. User Documentation for CVODE v5.7.0. Technical Report UCRL-SM-208108, LLNL, 2021.
- [34] A. C. Hindmarsh, R. Serban, and A. Collier. Example Programs for IDA v5.7.0. Technical Report UCRL-SM-208113, LLNL, 2021.
- [35] A. C. Hindmarsh, R. Serban, and D. R. Reynolds. Example Programs for CVODE v5.7.0. Technical report, LLNL, 2021. UCRL-SM-208110.
- [36] 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.
- [37] Seth R. Johnson, Andrey Prokopenko, and Katherine J. Evans. Automated fortran-c++ bindings for large-scale scientific applications. arXiv:1904.02546 [cs], 2019.

BIBLIOGRAPHY 383

[38] C. T. Kelley. *Iterative Methods for Solving Linear and Nonlinear Equations*. SIAM, Philadelphia, 1995.

- [39] X. S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.
- [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] X.S. Li, J.W. Demmel, J.R. Gilbert, L. Grigori, M. Shao, and I. Yamazaki. SuperLU Users' Guide. Technical Report LBNL-44289, Lawrence Berkeley National Laboratory, September 1999. http://crd.lbl.gov/~xiaoye/SuperLU/. Last update: August 2011.
- [42] Daniel R. Reynolds. Example Programs for ARKODE v4.7.0. Technical report, Southern Methodist University, 2021.
- [43] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. SIAM J. Sci. Comput., 14(2):461–469, 1993.
- [44] 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.
- [45] 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.
- [46] H. A. Van Der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. SIAM J. Sci. Stat. Comp., 13:631–644, 1992.

# Index

BIG_REAL, 30, 120, 127	FIDAPSET, 101
booleantype, 30	FIDAPSOL, 101
	FIDAREINIT, 102
CONSTR_VEC, 104	FIDARESFUN, 96
	FIDASETIIN, 103
data types	FIDASETRIN, 103
Fortran, 93	FIDASETVIN, 103
	FIDASOLVE, 102
eh_data, 74	FIDASPARSESETJAC, 100
error messages, 44	FIDASPILSETJAC, 101
redirecting, 44	FIDASPILSETPREC, 102
user-defined handler, 44, 74	FIDASPILSINIT, 98
DVD 4 intenference delle	FIDATOLREINIT, 103
FIDA interface module	fnvector_serial_mod, 141
interface to the IDABBDPRE module, 107–108	FSUNBANDLINSOLINIT, 263
optional input and output, 103	FSUNDENSELINSOLINIT, 261
rootfinding, 106 usage, 96–103	FSUNKLUINIT, 273
user-callable functions, 94–95	FSUNKLUREINIT, 273
user-supplied functions, 95	FSUNKLUSETORDERING, 274
fida_mod, 87	FSUNLAPACKBANDINIT, 268
FIDABANDSETJAC, 99	FSUNLAPACKDENSEINIT, 266
FIDABBDINIT, 107	fsunlinsol_band_mod, 263
FIDABBDOPT, 108	fsunlinsol_dense_mod, 260
FIDABBDREINIT, 108	fsunlinsol_klu_mod, 273
FIDABJAC, 99	fsunlinsol_pcg_mod, 319
FIDACOMMFN, 108	fsunlinsol_spbcgs_mod, 306
FIDADENSESETJAC, 99	fsunlinsol_spfgmr_mod, 298
FIDADJAC, 98	fsunlinsol_spgmr_mod, 291
FIDADLSINIT, 98	fsunlinsol_sptfqmr_mod, 312
FIDAEWT, 98	FSUNMASSBANDLINSOLINIT, 264
FIDAEWTSET, 98	FSUNMASSDENSELINSOLINIT, 261
FIDAFREE, 103	FSUNMASSKLUINIT, 273
FIDAGETDKY, 102	FSUNMASSKLUREINIT, 274
FIDAGETERRWEIGHTS, 103	FSUNMASSKLUSETORDERING, 274
FIDAGETESTLOCALERR, 106	FSUNMASSLAPACKBANDINIT, 268
FIDAGLOCFN, 108	FSUNMASSLAPACKDENSEINIT, 266
FIDAJTIMES, 100, 109	FSUNMASSPCGINIT, 319
FIDAJTSETUP, 100, 109	FSUNMASSPCGSETMAXL, 320
FIDALSINIT, 98	FSUNMASSPCGSETPRECTYPE, 320
FIDALSSETJAC, 100	FSUNMASSSPBCGSINIT, 306
FIDALSSETPREC, 101	FSUNMASSSPBCGSSETMAXL, 307
FIDAMALLOC, 98	FSUNMASSSPBCGSSETPRECTYPE, 307
FIDAMALLOC, 97	FSUNMASSSPFGMRINIT, 299

FSUNMASSSPFGMRSETGSTYPE, 299	IDA_BAD_DKY, 59
FSUNMASSSPFGMRSETMAXRS, 300	IDA_BAD_EWT, 41
FSUNMASSSPFGMRSETPRECTYPE, 300	IDA_BAD_K, 59
FSUNMASSSPGMRINIT, 291	IDA_BAD_T, 59
FSUNMASSSPGMRSETGSTYPE, 292	IDA_CONSTR_FAIL, 41, 43
FSUNMASSSPGMRSETMAXRS, 293	IDA_CONV_FAIL, 41, 43
FSUNMASSSPGMRSETPRECTYPE, 293	IDA_ERR_FAIL, 43
FSUNMASSSPTFQMRINIT, 312	IDA_FIRST_RES_FAIL, 41
FSUNMASSSPTFQMRSETMAXL, 314	IDA_ILL_INPUT, 36, 37, 40, 41, 43, 46, 47, 49, 50,
FSUNMASSSPTFQMRSETPRECTYPE, 313	55–58, 67, 73
FSUNMASSSUPERLUMTINIT, 282	IDA_LINESEARCH_FAIL, 41
FSUNMASSUPERLUMTSETORDERING, 282	IDA_LINIT_FAIL, 41, 43
fsunmatrix_band_mod, 225	IDA_LSETUP_FAIL, 41, 43
fsunmatrix_dense_mod, 219	IDA_LSOLVE_FAIL, 41, 43
fsunmatrix_sparse_mod, 232	IDA_MEM_FAIL, 36, 48, 66, 67
FSUNNEWTONINIT, 340	IDA_MEM_NULL, 36-38, 40, 41, 43, 44, 46-50, 55-59,
fsunnonlinsol_newton_mod, 340	61–68, 73
FSUNPCGINIT, 319	IDA_NO_MALLOC, 37, 38, 41, 73
FSUNPCGSETMAXL, 320	IDA_NO_RECOVERY, 41
FSUNPCGSETPRECTYPE, 320	IDA_NORMAL, 42
FSUNSPBCGSINIT, 306	IDA_ONE_STEP, 42
FSUNSPBCGSSETMAXL, 307	IDA_REP_RES_ERR, 43
FSUNSPECGSSETPRECTYPE, 307	IDA_RES_FAIL, 41, 43
FSUNSPEGNEINIT, 299	IDA_ROOT_RETURN, 43
FSUNSPFGMRSETGSTYPE, 299	IDA_RTFUNC_FAIL, 43, 75
	IDA_SUCCESS, 36–38, 40, 41, 43, 44, 46–50, 55–59,
FSUNSPFGMRSETMAXRS, 300	68, 73
FSUNSPFGMRSETPRECTYPE, 300	
FSUNSPGMRINIT, 291	IDA_TOO_MUCH_ACC, 43
FSUNSPGMRSETGSTYPE, 292	IDA_TOO_MUCH_WORK, 43
FSUNSPGMRSETMAXRS, 293	IDA_TSTOP_RETURN, 43
FSUNSPEMENTE 212	IDA VINIT 41
FSUNSPTFQMRINIT, 312	IDA_Y_INIT, 41
FSUNSPTFQMRSETMAXL, 313	IDA_YA_YDP_INIT, 41
FSUNSPTFQMRSETPRECTYPE, 313	IDABBDPRE preconditioner
FSUNSUPERLUMTINIT, 281	description, 81–82
FSUNSUPERLUMTSETORDERING, 282	optional output, 85–86
1-1f 1 1 141- 04	usage, 83–84
half-bandwidths, 84	user-callable functions, 84–85
header files, 31, 83	user-supplied functions, 82–83
ID UEC 104	IDABBDPrecGetNumGfnEvals, 86
ID_VEC, 104	IDABBDPrecGetWorkSpace, 85
IDA	IDABBDPrecInit, 84
motivation for writing in C, 1	IDABBDPrecReInit, 85
package structure, 26	IDACalcIC, 41
IDA linear solver interface	IDACreate, 36
IDALS, 39	IDAD1sGetLastFlag, 72
IDA linear solver interfaces, 27	IDAD1sGetNumJacEvals, 69
IDA linear solvers	IDAD1sGetNumRhsEvals, 69
header files, 31	IDAD1sGetReturnFlagName, 72
implementation details, 27–28	IDAD1sGetWorkspace, 69
NVECTOR compatibility, 29	IDADISJacFn, 77
selecting one, 39	IDAD1sSetJacFn, 51
ida/ida.h, 31	IDADIsSetLinearSolver, 40
ida/ida_ls.h, 31	IDAErrHandlerFn, 74

IDAEwtFn, 75	IDALS_PMEM_NULL, 85, 86
IDAFree, 35, 36	IDALS_SUCCESS, 39, 50-54, 69-72, 84-86
IDAGetActualInitStep, 64	IDALS_SUNLS_FAIL, 40, 52, 54
IDAGetConsistentIC, 67	IDALsJacFn, 75
IDAGetCurrentCj, 334	IDALsJacTimesSetupFn, 78
IDAGetCurrentOrder, 63	IDALsJacTimesVecFn, 77
IDAGetCurrentStep, 64	IDALsPrecSetupFn, 80
IDAGetCurrentTime, 64	IDALsPrecSolveFn, 79
IDAGetCurrentY, 335	IDAReInit, 73
IDAGetCurrentYp, 335	IDAResFn, 36, 53, 74
IDAGetDky, 58	IDARootFn, 75
IDAGetErrWeights, 65	IDARootInit, 42
IDAGetEstLocalErrors, 65	IDASetConstraints, 50
IDAGetIntegratorStats, 65	IDASetEpsLin, 54
IDAGetLastLinFlag, 72	IDASetErrFile, 44
IDAGetLastOrder, 63	IDASetErrHandlerFn, 44
IDAGetLastStep, 63	IDASetId, 49
IDAGetLinReturnFlagName, 72	IDASetIncrementFactor, 52
IDAGetLinWorkSpace, 68	IDASetInitStep, 47
IDAGetNonlinearSystemData, 3, 335	IDASetJacFn, 50
IDAGetNonlinSolvStats, 66	IDASetJacTimes, 52
IDAGetNumBacktrackOps, 67	IDASetJacTimesResFn, 53
IDAGetNumErrTestFails, 63	IDASetLinearSolutionScaling, 51
IDAGetNumGEvals, 68	IDASetLinearSolver, 34, 39, 75, 215
IDAGetNumJacEvals, 69	IDASetLineSearchOffIC, 57
IDAGetNumJtimesEvals, 71	IDASetLSNormFactor, 55
IDAGetNumJTSetupEvals, 71	IDASetMaxBacksIC, 57
IDAGetNumLinConvFails, 70	IDASetMaxConvFails, 48
IDAGetNumLinIters, 70	IDASetMaxErrTestFails, 48
IDAGetNumLinResEvals, 69	IDASetMaxNonlinIters, 48
IDAGetNumLinSolvSetups, 62	IDASetMaxNumItersIC, 56
IDAGetNumNonlinSolvConvFails, 66	IDASetMaxNumJacsIC, 56
IDAGetNumNonlinSolvIters, 66	IDASetMaxNumSteps, 46
IDAGetNumPrecEvals, 70	IDASetMaxNumStepsIC, 56
IDAGetNumPrecSolves, 71	IDASetMaxOrd, 46
IDAGetNumResEvals, 62	IDASetMaxStep, 47
IDAGetNumSteps, 62	IDASetNoInactiveRootWarn, 58
IDAGetReturnFlagName, 67	IDASetNonlinConvCoef, 48
IDAGetRootInfo, 68	IDASetNonlinConvCoefIC, 55
IDAGetTolScaleFactor, 64	IDASetNonLinearSolver, 40
IDAGetWorkSpace, 61	IDASetNonlinearSolver, 34, 40
IDAInit, 36, 73	IDASetPreconditioner, 53, 54
IDALS linear solver interface	IDASetRootDirection, 58
convergence test, 54	IDASetStepToleranceIC, 57
Jacobian approximation used by, 50, 51	IDASetStopTime, 47
memory requirements, 68	IDASetSuppressAlg, 49
optional input, 50–55	IDASetUserData, 46
optional output, 68–72	IDASolve, 34, 42
preconditioner setup function, 53, 80	IDASpilsGetLastFlag, 72
preconditioner setup function, 53, 60 preconditioner solve function, 53, 79	IDASpilsGetNumConvFails, 70
IDALS_ILL_INPUT, 39, 51–54, 84	IDASpilsGetNumJtimesEvals, 71
IDALS_LMEM_NULL, 50-54, 69-72, 84, 85	IDASpilsGetNumJTSetupEvals, 71
IDALS_MEM_FAIL, 40, 84	IDASpilsGetNumLinIters, 70
IDALS_MEM_NULL, 39, 50-54, 69-72, 84-86	IDASpilsGetNumPrecEvals, 70
,,,,,,,,,,,	

IDASpilsGetNumPrecSolves, 71	IDALS linear solver interface, 68
IDASpilsGetNumRhsEvals, 69	
IDASpilsGetReturnFlagName, 72	$ exttt{NVCloneVectorArray}, 128$
IDASpilsGetWorkspace, 69	N_VCloneVectorArray_OpenMP, 148
IDASpilsJacTimesSetupFn, 79	N_VCloneVectorArray_OpenMPDEV, 189
IDASpilsJacTimesVecFn, 78	N_VCloneVectorArray_Parallel, 143
IDASpilsPrecSetupFn, 81	$ exttt{N_VCloneVectorArray_ParHyp}, 158$
IDASpilsPrecSolveFn, 80	N_VCloneVectorArray_Petsc, 162
IDASpilsSetEpsLin, 54	N_VCloneVectorArray_Pthreads, 154
IDASpilsSetIncrementFactor, 53	N_VCloneVectorArray_Serial, 137
IDASpilsSetJacTimes, 52	N_VCloneVectorArrayEmpty, 128
IDASpilsSetLinearSolver, 40	N_VCloneVectorArrayEmpty_OpenMP, 148
IDASpilsSetPreconditioner, 54	N_VCloneVectorArrayEmpty_OpenMPDEV, 189
IDASStolerances, 37	N_VCloneVectorArrayEmpty_Parallel, 143
IDASVtolerances, 37	N_VCloneVectorArrayEmpty_ParHyp, 158
IDAWFtolerances, 37	N_VCloneVectorArrayEmpty_Petsc, 162
INIT_STEP, 104	N_VCloneVectorArrayEmpty_Pthreads, 154
IOUT, 103, 105	N_VCloneVectorArrayEmpty_Serial, 137
itask, 42	N_VCopyFromDevice_Cuda, 167
,	N_VCopyFromDevice_Hip, 173
Jacobian approximation function	N_VCopyFromDevice_OpenMPDEV, 190
band	N_VCopyFromDevice_Raja, 179
use in FIDA, 99	N_VCopyFromDevice_Sycl, 183
dense	N_VCopyOps, 128
use in FIDA, 98	N_VCopyToDevice_Cuda, 167
difference quotient, 50	N_VCopyToDevice_Hip, 173
Jacobian times vector	N_VCopyToDevice_OpenMPDEV, 190
alternative-res, 53	N_VCopyToDevice_Raja, 179
difference quotient, 51	
increment, 52	N_VCopyToDevice_Sycl, 183
use in FIDA, 100	N_VDestroyVectorArray, 128
user-supplied, 51, 77–78	N_VDestroyVectorArray_OpenMP, 148
Jacobian-vector setup	N_VDestroyVectorArray_OpenMPDEV, 189
user-supplied, 78–79	N_VDestroyVectorArray_Parallel, 143
sparse	N_VDestroyVectorArray_ParHyp, 158
use in FIDA, 99	N_VDestroyVectorArray_Petsc, 162
user-supplied, 50, 75–77	N_VDestroyVectorArray_Pthreads, 154
11 , ,	N_VDestroyVectorArray_Serial, 138
Linear solution scaling function	N_Vector, 31, 113, 130
user-supplied, $51$	N_VEnableConstVectorArray_Cuda, 168
LS_OFF_IC, 104	N_VEnableConstVectorArray_Hip, 174
	N_VEnableConstVectorArray_ManyVector, 197
MAX_CONVFAIL, 104	N_VEnableConstVectorArray_MPIManyVector, 202
MAX_ERRFAIL, 104	N_VEnableConstVectorArray_OpenMP, 150
MAX_NITERS, 104	N_VEnableConstVectorArray_OpenMPDEV, 191
MAX_NITERS_IC, 104	N_VEnableConstVectorArray_Parallel, 145
MAX_NJE_IC, 104	N_VEnableConstVectorArray_ParHyp, 160
MAX_NSTEPS, 104	${\tt N\_VEnableConstVectorArray\_Petsc}, 163$
MAX_NSTEPS_IC, 104	${\tt N\_VEnableConstVectorArray\_Pthreads}, 156$
$\mathtt{MAX\_ORD},\ 104$	$ exttt{N_VEnableConstVectorArray_Raja}, 180$
MAX_STEP, 104	$ exttt{N\_VEnableConstVectorArray\_Serial}, 139$
maxord, 73	N_VEnableConstVectorArray_Sycl, 185
memory requirements	N_VEnableDotProdMulti_Cuda, 168
IDA solver, 61	${ t N\_VEnableDotProdMulti\_Hip}, 174$
IDABBDERE preconditioner 85	N VEnableDotProdMulti ManyVector 196

${\tt N\_VEnableDotProdMulti\_MPIManyVector}, \underline{201}$	${\tt N\_VEnableLinearCombinationVectorArray\_Sycl},$
N_VEnableDotProdMulti_OpenMP, 150	185
N_VEnableDotProdMulti_OpenMPDEV, 191	N_VEnableLinearSumVectorArray_Cuda, 168
N_VEnableDotProdMulti_Parallel, 144	N_VEnableLinearSumVectorArray_Hip, 174
N_VEnableDotProdMulti_ParHyp, 159	${\tt N\_VEnableLinearSumVectorArray\_ManyVector}, 197$
N_VEnableDotProdMulti_Petsc, 163	${\tt N\_VEnableLinearSumVectorArray\_MPIManyVector},$
N_VEnableDotProdMulti_Pthreads, 155	202
N_VEnableDotProdMulti_Serial, 139	${\tt N\_VEnableLinearSumVectorArray\_OpenMP}, 150$
N_VEnableFusedOps_Cuda, 168	${\tt N\_VEnableLinearSumVectorArray\_OpenMPDEV}, \underline{191}$
N_VEnableFusedOps_Hip, 174	${\tt N\_VEnableLinearSumVectorArray\_Parallel}, {\tt 145}$
N_VEnableFusedOps_ManyVector, 196	N_VEnableLinearSumVectorArray_ParHyp, 159
N_VEnableFusedOps_MPIManyVector, 201	N_VEnableLinearSumVectorArray_Petsc, 163
N_VEnableFusedOps_OpenMP, 149	${\tt N\_VEnableLinearSumVectorArray\_Pthreads}, 155$
N_VEnableFusedOps_OpenMPDEV, 190	N_VEnableLinearSumVectorArray_Raja, 180
N_VEnableFusedOps_Parallel, 144	N_VEnableLinearSumVectorArray_Serial, 139
N_VEnableFusedOps_ParHyp, 159	N_VEnableLinearSumVectorArray_Sycl, 185
N_VEnableFusedOps_Petsc, 162	N_VEnableScaleAddMulti_Cuda, 168
N_VEnableFusedOps_Pthreads, 154	N_VEnableScaleAddMulti_Hip, 174
N_VEnableFusedOps_Raja, 180	N_VEnableScaleAddMulti_ManyVector, 196
N_VEnableFusedOps_Serial, 138	N_VEnableScaleAddMulti_MPIManyVector, 201
	N_VEnableScaleAddMulti_OpenMP, 149
N_VEnableFusedOps_Sycl, 184	N_VEnableScaleAddMulti_OpenMPDEV, 190
N_VEnableLinearCombination_Cuda, 168	N_VEnableScaleAddMulti_Parallel, 144
N_VEnableLinearCombination_Hip, 174	N_VEnableScaleAddMulti_ParHyp, 159
N_VEnableLinearCombination_ManyVector, 196	N_VEnableScaleAddMulti_Petsc, 163
N_VEnableLinearCombination_MPIManyVector, 20	N_VEnableScaleAddMulti_Pthreads, 155
N_VEHADIELIHEAI COMDINACIOH_OPEHNF, 149	N_VEnableScaleAddMulti_Raja, 180
N_VEnableLinearCombination_OpenMPDEV, 190	N_VEnableScaleAddMulti_Serial, 139
N_VEnableLinearCombination_Parallel, 144	N_VEnableScaleAddMulti_Sycl, 185
N_VEnableLinearCombination_ParHyp, 159	N_VEnableScaleAddMultiVectorArray_Cuda, 169
$N_VEnableLinearCombination\_Petsc, 162$	N_VEnableScaleAddMultiVectorArray_Hip, 175
${ t N\_VEnable Linear Combination\_Pthreads}, 155$	N_VEnableScaleAddMultiVectorArray_OpenMP, 151
${\tt N\_VEnableLinearCombination\_Raja},180$	N_VEnableScaleAddMultiVectorArray_OpenMPDEV,
N_VEnableLinearCombination_Serial, 138	191
N_VEnableLinearCombination_Sycl, 185	N_VEnableScaleAddMultiVectorArray_Parallel,
${\tt N\_VEnableLinearCombinationVectorArray\_Cuda},$	145
169	N_VEnableScaleAddMultiVectorArray_ParHyp, 160
${\tt N\_VEnableLinearCombinationVectorArray\_Hip}, 1$	₩VEnableScaleAddMultiVectorArray_Petsc, 164
N_VEnableLinearCombinationVectorArray_OpenM	N_VEnableScaleAddMultiVectorArray_Pthreads,
151	156
${\tt N\_VEnableLinearCombinationVectorArray\_OpenM}$	RPDVEnableScaleAddMultiVectorArray_Raja, 181
192	N_VEnableScaleAddMultiVectorArray_Serial, 140
${\tt N\_VEnableLinearCombinationVectorArray\_Paral}$	
146	N_VEnableScaleVectorArray_Cuda, 168
${\tt N\_VEnableLinearCombinationVectorArray\_ParHy}$	
160	N_VEnableScaleVectorArray_ManyVector, 197
N_VEnableLinearCombinationVectorArray_Petsc	,N_VEnableScaleVectorArray_MPIManyVector, 202
164	N_VEnableScaleVectorArray_OpenMP, 150
N_VEnableLinearCombinationVectorArray_Pthre	
156	N_VEnableScaleVectorArray_Parallel, 145
N_VEnableLinearCombinationVectorArray_Raja,	•
181	N_VEnableScaleVectorArray_Petsc, 163
N_VEnableLinearCombinationVectorArray_Seria	· · · · · · · · · · · · · · · · · · ·
140	N_VEnableScaleVectorArray_Raja, 180

N_VEnableScaleVectorArray_Serial, 139	N_VMake_Cuda, 166
N_VEnableScaleVectorArray_Sycl, 185	N_VMake_Hip, 172
N_VEnableWrmsNormMaskVectorArray_Cuda, 169	N_VMake_MPIManyVector, 200
N_VEnableWrmsNormMaskVectorArray_Hip, 175	N_VMake_MPIPlusX, 203
${\tt N\_VEnableWrmsNormMaskVectorArray\_ManyVect$	,N_VMake_OpenMP, 148
197	N_VMake_OpenMPDEV, 189
${\tt N\_VEnableWrmsNormMaskVectorArray\_MPIManyVectorArray\_MPIMANAMANAMANAMANAMANAMANAMANAMANAMANAMAN$	tto.WMake_Parallel, 143
202	N_VMake_ParHyp, 158
${\tt N\_VEnableWrmsNormMaskVectorArray\_OpenMP}, 150$	N_VMake_Petsc, 161
${\tt N\_VEnableWrmsNormMaskVectorArray\_OpenMPDEV},$	N_VMake_Pthreads, 153
191	N_VMake_Raja, 179
${\tt N\_VEnableWrmsNormMaskVectorArray\_Parallel, 1}$	45_VMake_Serial, 137
N_VEnableWrmsNormMaskVectorArray_ParHyp, 160	N_VMake_Syc1, 182
N_VEnableWrmsNormMaskVectorArray_Petsc, 163	
${\tt N\_VEnableWrmsNormMaskVectorArray\_Pthreads}, 1$	
N_VEnableWrmsNormMaskVectorArray_Serial, 140	<del>-</del>
N_VEnableWrmsNormVectorArray_Cuda, 169	N_VMakeManaged_Raja, 179
N_VEnableWrmsNormVectorArray_Hip, 175	N_VMakeManaged_Sycl, 182
N_VEnableWrmsNormVectorArray_ManyVector, 197	<u> </u>
N_VEnableWrmsNormVectorArray_MPIManyVector,	
202	N_VNew_Hip, 172
N_VEnableWrmsNormVectorArray_OpenMP, 150	N_VNew_ManyVector, 194
N_VEnableWrmsNormVectorArray_OpenMPDEV, 191	N_VNew_MPIManyVector, 199, 200
N_VEnableWrmsNormVectorArray_Parallel, 145	N_VNew_OpenMP, 148
N_VEnableWrmsNormVectorArray_ParHyp, 160	N_VNew_OpenMPDEV, 189
N_VEnableWrmsNormVectorArray_Petsc, 163	N_VNew_Parallel, 142
N_VEnableWrmsNormVectorArray_Pthreads, 156	N_VNew_Pthreads, 153
N_VEnableWrmsNormVectorArray_Serial, 139	N_VNew_Raja, 178
N_VGetArrayPointer_MPIPlusX, 204	N_VNew_SensWrapper, 333
N_VGetDeviceArrayPointer_Cuda, 165	N_VNew_Serial, 137
N_VGetDeviceArrayPointer_Hip, 172	N_VNew_Sycl, 182
N_VGetDeviceArrayPointer_OpenMPDEV, 189	N_VNewEmpty, 128
N_VGetDeviceArrayPointer_Raja, 178	N_VNewEmpty_Cuda, 166
N_VGetDeviceArrayPointer_Sycl, 183	N_VNewEmpty_Hip, 172
N_VGetHostArrayPointer_Cuda, 165	N_VNewEmpty_OpenMP, 148
N_VGetHostArrayPointer_Hip, 172	N_VNewEmpty_OpenMPDEV, 189
N_VGetHostArrayPointer_OpenMPDEV, 189	N_VNewEmpty_Parallel, 142
N_VGetHostArrayPointer_Raja, 178	N_VNewEmpty_ParHyp, 158
N_VGetHostArrayPointer_Sycl, 183	N_VNewEmpty_Petsc, 161
N_VGetLocalLength_Parallel, 143	N_VNewEmpty_Pthreads, 153
N_VGetLocalVector_MPIPlusX, 204	N_VNewEmpty_Raja, 179
N_VGetNumSubvectors_ManyVector, 195	N_VNewEmpty_SensWrapper, 332
N_VGetNumSubvectors_MPIManyVector, 201	N_VNewEmpty_Serial, 137
N_VGetSubvector_ManyVector, 195	N_VNewEmpty_Sycl, 183
N_VGetSubvector_MPIManyVector, 200	N_VNewManaged_Cuda, 166
N_VGetSubvectorArrayPointer_ManyVector, 195	N_VNewManaged_Hip, 172
N_VGetSubvectorArrayPointer_MPIManyVector, 2	ON_VNewManaged_Raja, 179
N_VGetVector_ParHyp, 158	N_VNewManaged_Syc1, 182
N_VGetVector_Petsc, 161	N_VNewWithMemHelp_Cuda, 166
N_VGetVector_Trilinos, 193	N_VNewWithMemHelp_Raja, 178
N_VIsManagedMemory_Cuda, 165	N_VNewWithMemHelp_Syc1, 183
N_VIsManagedMemory_Hip, 172	N_VPrint_Cuda, 167
N_VIsManagedMemory_Raja, 178	N_VPrint_Hip, 173
N_VIsManagedMemory_Syc1, 184	N_VPrint_OpenMP, 149

N_VPrint_OpenMPDEV, 190	NV_LENGTH_S, 136
N_VPrint_Parallel, 143	NV_LOCLENGTH_P, 141
N_VPrint_ParHyp, 158	NV_NUM_THREADS_OMP, 147
N_VPrint_Petsc, 162	NV_NUM_THREADS_PT, 152
N_VPrint_Pthreads, 154	NV_OWN_DATA_OMP, 147
N_VPrint_Raja, 179	NV_OWN_DATA_OMPDEV, 188
N_VPrint_Serial, 138	NV_OWN_DATA_P, 141
N_VPrint_Syc1, 184	NV_OWN_DATA_PT, 152
N_VPrintFile_Cuda, 167	NV_OWN_DATA_S, 136
N_VPrintFile_Hip, 173	NVECTOR module, 113
N_VPrintFile_OpenMP, 149	nvector_openmp_mod, 151
N_VPrintFile_OpenMPDEV, 190	nvector_pthreads_mod, 157
N_VPrintFile_Parallel, 144	
N_VPrintFile_ParHyp, 159	optional input
N_VPrintFile_Petsc, 162	generic linear solver interface, 50–55
N_VPrintFile_Pthreads, 154	initial condition calculation, 55–57
N_VPrintFile_Raja, 179	iterative linear solver, 53–55
<u> </u>	matrix-based linear solver, 50–51
N_VPrintFile_Serial, 138	matrix-free linear solver, 51–53
N_VPrintFile_Sycl, 184	rootfinding, 57–58
N_VSetArrayPointer_MPIPlusX, 204	solver, 44–50
N_VSetCudaStream_Cuda, 167	optional output
N_VSetDeviceArrayPointer_Cuda, 165	band-block-diagonal preconditioner, 85–86
N_VSetDeviceArrayPointer_Raja, 178	generic linear solver interface, 68–72
N_VSetDeviceArrayPointer_Syc1, 183	initial condition calculation, 67
N_VSetHostArrayPointer_Cuda, 165	interpolated solution, 58
N_VSetHostArrayPointer_Raja, 178	
N_VSetHostArrayPointer_Syc1, 183	solver, 61–67
N_VSetKernelExecPolicy_Cuda, 167	version, 59–61
N_VSetKernelExecPolicy_Hip, 173	portability, 30
N_VSetKernelExecPolicy_Sycl, 184	Fortran, 93
${\tt N\_VSetSubvectorArrayPointer\_ManyVector}, 195$	Preconditioner setup routine
${\tt N\_VSetSubvectorArrayPointer\_MPIManyVector}, \ 2\\$	use in FIDA, 101
NLCONV_COEF, 104	Preconditioner solve routine
NLCONV_COEF_IC, 104	use in FIDA, 101
NV_COMM_P, 142	
NV_CONTENT_OMP, 147	preconditioning
NV_CONTENT_OMPDEV, 188	advice on, 23, 27
NV_CONTENT_P, 141	band-block diagonal, 81
NV_CONTENT_PT, 152	setup and solve phases, 27
NV_CONTENT_S, 136	user-supplied, 53–54, 79, 80
NV_DATA_DEV_OMPDEV, 188	RCONST, 30
NV_DATA_HOST_OMPDEV, 188	
NV_DATA_OMP, 147	realtype, 30
NV_DATA_P, 141	reinitialization, 73
NV_DATA_PT, 152	residual function, 74
NV_DATA_S, 136	Rootfinding, 23, 34, 42, 106
NV_GLOBLENGTH_P, 141	ROUT, 103, 105
NV_Ith_OMP, 147	CM COIC D 999
	SM_COLS_B, 222
NV_Ith_P, 142 NV_I+b_PT_153	SM_COLS_D, 217
NV_Ith_PT, 153	SM_COLUMN_B, 77, 222
NV_Ith_S, 136	SM_COLUMN_D, 76, 217
NV_LENGTH_OMP, 147	SM_COLUMN_ELEMENT_B, 77, 222
NV_LENGTH_OMPDEV, 188	SM_COLUMNS_B, 222
NV_LENGTH_PT, 152	SM_COLUMNS_D, 217

SM_COLUMNS_S, 229	SUNLinearSolver, 245, 253
SM_CONTENT_B, 220	SUNLinearSolver module, 245
SM_CONTENT_D, 216	SUNLINEARSOLVER_DIRECT, 76, 247, 256
SM_CONTENT_S, 227	SUNLINEARSOLVER_ITERATIVE, 247, 257
SM_DATA_B, 222	SUNLINEARSOLVER_MATRIX_ITERATIVE, 247, 257
SM_DATA_D, 217	sunlinsol/sunlinsol_band.h, 31
SM_DATA_S, 229	sunlinsol/sunlinsol_dense.h, 31
SM_ELEMENT_B, 77, 222	sunlinsol/sunlinsol_klu.h, 31
SM_ELEMENT_D, 76, 217	sunlinsol/sunlinsol_lapackband.h, 31
SM_INDEXPTRS_S, 229	sunlinsol/sunlinsol_lapackdense.h, 31
SM_INDEXVALS_S, 229	sunlinsol/sunlinsol_pcg.h, 32
SM_LBAND_B, 222	sunlinsol/sunlinsol_spbcgs.h, 32
SM_LDATA_B, 222	sunlinsol/sunlinsol_spfgmr.h, 32
SM_LDATA_D, 217	sunlinsol/sunlinsol_spgmr.h, 32
SM_LDIM_B, 222	sunlinsol/sunlinsol_sptfqmr.h, 32
SM_NNZ_S, 77, 229	sunlinsol/sunlinsol_superlumt.h, 31
SM_NP_S, 229	SUNLinSol_Band, 39, 262
SM_ROWS_B, 222	SUNLinSol_cuSolverSp_batchQR, 284
SM_ROWS_D, 217	SUNLinSol_cuSolverSp_batchQR_GetDescription,
SM_ROWS_S, 229	285
SM_SPARSETYPE_S, 229	SUNLinSol_cuSolverSp_batchQR_SetDescription,
SM_SUBAND_B, 222	285
SM_UBAND_B, 222	SUNLinSol_Dense, 39, 260
SMALL_REAL, 30	SUNLinSol_KLU, 39, 270
step size bounds, 47	SUNLinSol_KLUReInit, 271
STEP_TOL_IC, 104	SUNLinSol_KLUSetOrdering, 273
STOP_TIME, 104	SUNLinSol_LapackBand, 39, 267
SUNBandMatrix, 33, 223	SUNLinSol_LapackDense, 39, 265
SUNBandMatrix_Cols, 224	SUNLinSol_MagmaDense, 286
SUNBandMatrix_Column, 225	SUNLinSol_PCG, 39, 316, 319, 320
SUNBandMatrix_Columns, 224	SUNLinSol_PCGSetMaxl, 318
SUNBandMatrix_Data, 224	SUNLinSol_PCGSetPrecType, 317
SUNBandMatrix_LDim, 224	SUNLinSol_SPBCGS, 39, 303, 306
SUNBandMatrix_LowerBandwidth, 224	SUNLinSol_SPBCGSSetMax1, 304
SUNBandMatrix_Print, 223	SUNLinSol_SPBCGSSetPrecType, 304
SUNBandMatrix_Rows, 223	SUNLinSol_SPFGMR, 39, 295, 299
SUNBandMatrix_StoredUpperBandwidth, 224	SUNLinSol_SPFGMRSetMaxRestarts, 297
SUNBandMatrix_UpperBandwidth, 224	SUNLinSol_SPFGMRSetPrecType, 296, 297
SUNBandMatrixStorage, 223	SUNLinSol_SPGMR, 39, 288, 291, 292
SUNDenseMatrix, 33, 217	SUNLinSol_SPGMRSetMaxRestarts, 290
SUNDenseMatrix_Cols, 218	SUNLinSol_SPGMRSetPrecType, 289
SUNDenseMatrix_Column, 218	SUNLinSol_SPTFQMR, 39, 309, 312, 313
SUNDenseMatrix_Columns, 218	SUNLinSol_SPTFQMRSetMax1, 311
SUNDenseMatrix_Data, 218	SUNLinSol_SPTFQMRSetPrecType, 310
SUNDenseMatrix_LData, 218	SUNLinSol_SuperLUDIST, 277
SUNDenseMatrix_Print, 218	SUNLinSol_SuperLUDIST_GetBerr, 277
SUNDenseMatrix_Rows, 218	SUNLinSol_SuperLUDIST_GetGridinfo, 277
sundials/sundials_linearsolver.h, 245	SUNLinSol_SuperLUDIST_GetLUstruct, 278
sundials_nonlinearsolver.h, 31	SUNLinSol_SuperLUDIST_GetScalePermstruct, 278
sundials_nvector.h, 31	SUNLinSol_SuperLUDIST_GetSOLVEstruct, 278
sundials_types.h, 30, 31	SUNLinSol_SuperLUDIST_GetSuperLUOptions, 278
SUNDIALSGetVersion, 59	SUNLinSol_SuperLUDIST_GetSuperLUStat, 278
SUNDIALSGetVersionNumber, 61	SUNLinSol_SuperLUMT, 39, 280
sunindextype, 30	SUNLinSol_SuperLUMTSetOrdering, 282

SUNLinSolFree, 35, 246, 248	SUNNonlinSolInitialize, 324
SUNLinSolGetID, 246, 247	SUNNonlinSolLSetupFn, 326
SUNLinSolGetType, 246, 247	SUNNonlinSolNewEmpty, 334
SUNLinSolInitialize, 246, 247	SUNNonlinSolSetConvTestFn, 326
SUNLinSolLastFlag, 250	SUNNonlinSolSetInfoFile_Newton, 339
SUNLinSolNewEmpty, 255	SUNNonlinSolSetLSolveFn, 326
SUNLinSolNumIters, 250	SUNNonlinSolSetMaxIters, 327
SUNLinSolResNorm, 250	SUNNonlinSolSetPrintLevel_Newton, 339
SUNLinSolSetATimes, 247-249, 257	SUNNonlinSolSetSysFn, 325
SUNLinSolSetInfoFile_PCG, 318	SUNNonlinSolSetup, 324
SUNLinSolSetInfoFile_SPBCGS, 305	SUNNonlinSolSolve, 324
SUNLinSolSetInfoFile_SPFGMR, 297	SUNSparseFromBandMatrix, 230
SUNLinSolSetInfoFile_SPGMR, 290	SUNSparseFromDenseMatrix, 230
SUNLinSolSetInfoFile_SPTFQMR, 311	SUNSparseMatrix, 33, 229
SUNLinSolSetPreconditioner, 249	SUNSparseMatrix_Columns, 231
SUNLinSolSetPrintLevel_PCG, 318	SUNSparseMatrix_Data, 231
SUNLinSolSetPrintLevel_SPBCGS, 305	SUNSparseMatrix_IndexPointers, 232
SUNLinSolSetPrintLevel_SPFGMR, 298	SUNSparseMatrix_IndexValues, 232
SUNLinSolSetPrintLevel_SPGMR, 290	SUNSparseMatrix_NNZ, 77, 231
SUNLinSolSetPrintLevel_SPTFQMR, 311	SUNSparseMatrix_NP, 231
SUNLinSolSetScalingVectors, 249	SUNSparseMatrix_Print, 231
SUNLinSolSetup, 246, 248, 257	SUNSparseMatrix_Realloc, 230
	SUNSparseMatrix_Reallocate, 230
SUNLinSolSolve, 246, 248	SUNSparseMatrix_Rows, 231
SUNLinSolSpace, 251 SUNMatCopyOps, 212	SUNSparseMatrix_SparseType, 231
	SUPPRESS_ALG, 104
SUNMatDestroy, 35	5011111155_11114, 101
SUNMatNewEmpty, 212	tolerances, 20, 38, 75
SUNMatrix, 209, 214	, , ,
SUNMatrix module, 209	UNIT_ROUNDOFF, 30
SUNMatrix_SLUNRloc, 233	User main program
SUNMatrix_SLUNRloc_OwnData, 234	FIDA usage, 96
SUNMatrix_SLUNRloc_Print, 234	FIDABBD usage, 107
SUNMatrix_SLUNRloc_ProcessGrid, 234	IDA usage, 32
SUNMatrix_SLUNRloc_SuperMatrix, 234	IDABBDPRE usage, 83
SUNMemory, 345	${\tt user\_data}, 46, 74, 75, 82$
SUNMemory module, 345	
SUNMemoryHelper, 345	weighted root-mean-square norm, 20
SUNMemoryType, 345	
SUNNonlinearSolver, 31, 323	
SUNNonlinearSolver module, 323	
SUNNONLINEARSOLVER_FIXEDPOINT, 324	
SUNNONLINEARSOLVER_ROOTFIND, 324	
SUNNonlinSol_Newton, 338	
SUNNonlinSol_NewtonSens, 338	
SUNNonlinSol_PetscSNES, 342	
SUNNonlinSolFree, 35, 325	
SUNNonlinSolGetCurIter, 327	
SUNNonlinSolGetNumConvFails, 328	
SUNNonlinSolGetNumIters, 327	
SUNNonlinSolGetPetscError_PetscSNES, 342	
SUNNonlinSolGetSNES_PetscSNES, 342	
SUNNonlinSolGetSysFn_Newton, 338	
SUNNonlinSolGetSysFn_PetscSNES, 343	
SUNNonlinSolGetType, 324	