User Documentation for CVODE v5.7.0 (SUNDIALS v5.7.0)

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

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

February 2, 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

Li	ist of Tables	ix
Li	ist of Figures	xi
1	Introduction1.1 Historical Background1.2 Changes from previous versions1.3 Reading this User Guide1.4 SUNDIALS Release License	1 2 16 17
2	Mathematical Considerations 2.1 IVP solution 2.2 IVPs with constraints 2.3 Preconditioning . 2.4 BDF stability limit detection 2.5 Rootfinding .	19 19 23 24 25 26
3	Code Organization3.1 SUNDIALS organization3.2 CVODE organization	27 27 28
4	Using CVODE 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 Preconditioner modules	31 31 32 33 34 37 80 89
5	Using CVODE for Fortran Applications 5.1 CVODE Fortran 2003 Interface Module	97 97 103
6	CVODE Features for GPU Accelerated Computing 6.1 SUNDIALS GPU Programming Model	
7	Description of the NVECTOR module 7.1 The NVECTOR API	$\frac{144}{146}$

	7.5	The NVECTOR_OPENMP implementation	156
		*	
	7.6	The NVECTOR_PTHREADS implementation	
	7.7	The NVECTOR_PARHYP implementation	
	7.8	The NVECTOR_PETSC implementation	
	7.9	The NVECTOR_CUDA implementation	
		The NVECTOR_HIP implementation	
		The NVECTOR_RAJA implementation	
	7.12	The NVECTOR_SYCL implementation $\ \ldots \ \ldots \ \ldots \ \ldots \ \ldots$	191
	7.13	The NVECTOR_OPENMPDEV implementation \hdots	197
	7.14	The NVECTOR_TRILINOS implementation	202
	7.15	The NVECTOR_MANYVECTOR implementation	203
		The NVECTOR_MPIMANYVECTOR implementation	
	7.17	The NVECTOR_MPIPLUSX implementation	213
		NVECTOR Examples	
0	ъ	· · · · · · · · · · · · · · · · · · ·	010
8	8.1	cription of the SUNMatrix module	219
	-	The SUNMatrix API	
	8.2	SUNMatrix functions used by CVODE	
	8.3	The SUNMatrix_Dense implementation	
	8.4	The SUNMatrix_Band implementation	
	8.5	The SUNMatrix_Sparse implementation	
	8.6	The SUNMatrix_SLUNRloc implementation	
	8.7	The SUNMatrix_cuSparse implementation	
	8.8	The SUNMATRIX_MAGMADENSE implementation	250
9	Desc	cription of the SUNLinearSolver module	255
	9.1	The SUNLinearSolver API	256
	9.2	Compatibility of SUNLinearSolver modules	
	9.3	Implementing a custom SUNLinearSolver module	
	9.4	CVODE SUNLinearSolver interface	
	9.5	The SUNLinearSolver_Dense implementation	
	9.6	The SUNLinearSolver_Band implementation	
	9.7	The SUNLinearSolver_LapackDense implementation	
	9.7		
	9.0	The SUNLinearSolver_KLU implementation	
		The SUNLinearSolver_SuperLUDIST implementation	
		The SUNLinearSolver_SuperLUMT implementation	
		The SUNLinearSolver_cuSolverSp_batchQR implementation	
		The SUNLinearSolver_MagmaDense implementation	
		The SUNLinearSolver_SPGMR implementation	297
		The SUNLinearSolver_SPFGMR implementation	305
		The SUNLinearSolver_SPBCGS implementation	312
		The SUNLinearSolver_SPTFQMR implementation	319
	9.18	The SUNLinear Solver_PCG implementation	325
	9.19	SUNLinearSolver Examples	331
10) Des	cription of the SUNNonlinearSolver module	333
_ 0		The SUNNonlinear Solver API	333
		CVODE SUNNonlinear Solver interface	344
		The SUNNonlinear Solver Newton implementation	346
		The SUNNonlinear Solver-Fixed Point implementation	
		The SUNNonlinear Solver PetscSNES implementation	350 356
	111.1	THE THE PRODUCTION OF THE PROTECT OF	

11	Description of the SUNMemory module	359
	11.1 The SUNMemoryHelper API	359
	11.2 The SUNMemoryHelper_Cuda implementation	
	11.3 The SUNMemoryHelper_Hip implementation	
	11.4 The SUNMemoryHelper_Sycl implementation	
\mathbf{A}	SUNDIALS Package Installation Procedure	369
	A.1 CMake-based installation	370
	A.2 Building and Running Examples	
	A.3 Configuring, building, and installing on Windows	
	A.4 Installed libraries and exported header files	
В	CVODE Constants	391
	B.1 CVODE input constants	391
	B.2 CVODE output constants	
\mathbf{C}	SUNDIALS Release History	395
Bi	ibliography	397
In	dex	401

List of Tables

4.1	SUNDIALS linear solver interfaces and vector implementations that can be used for each.	37
4.2	Optional inputs for CVODE and CVLS	47
4.3	Optional outputs from CVODE, CVLS, and CVDIAG	64
5.1	Summary of Fortran 2003 interfaces for shared SUNDIALS modules	98
5.2	C/Fortran 2003 Equivalent Types	99
5.3	Keys for setting FCVODE optional inputs	114
5.4	Description of the FCVODE optional output arrays IOUT and ROUT	115
	The A Court of the	
6.1	List of Sundials GPU Enabled Modules	122
7.1	Vector Identifications associated with vector kernels supplied with SUNDIALS	140
7.2	List of vector functions usage by CVODE code modules	
1.2	List of vector functions using by event code modules	110
8.1	Description of the SUNMatrix return codes	222
8.2	Identifiers associated with matrix kernels supplied with SUNDIALS	
8.3	SUNDIALS matrix interfaces and vector implementations that can be used for each	223
8.4	List of matrix functions usage by CVODE code modules	225
9.1	1	262
9.2	SUNDIALS matrix-based linear solvers and matrix implementations that can be used for	
	each	265
9.3	List of linear solver function usage in the CVLS interface	268
10.1	Description of the SUNNonlinearSolver return codes	340
10.1	Description of the SoundiffhearSolver return codes	340
A.1	SUNDIALS libraries and header files	384
C.1	Release History	395

List of Figures

3.1	High-level diagram of the SUNDIALS suite	7
3.2	Directory structure of the SUNDIALS source tree	8
3.3	Overall structure diagram of the CVODE package	9
	Diagram of the storage for a SUNMATRIX_BAND object	
8.2	Diagram of the storage for a compressed-sparse-column matrix	8
A.1	Initial <i>ccmake</i> configuration screen	1
A.2	Changing the <i>instdir</i>	2

Chapter 1

Introduction

CVODE is part of a software family called SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic equation Solvers [36]. This suite consists of CVODE, ARKODE, KINSOL, and IDA, and variants of these with sensitivity analysis capabilities.

1.1 Historical Background

FORTRAN solvers for ODE initial value problems are widespread and heavily used. Two solvers that have been written at LLNL in the past are VODE [13] and VODPK [16]. VODE is a general purpose solver that includes methods for both stiff and nonstiff systems, and in the stiff case uses direct methods (full or banded) for the solution of the linear systems that arise at each implicit step. Externally, VODE is very similar to the well known solver LSODE [47]. VODPK is a variant of VODE that uses a preconditioned Krylov (iterative) method, namely GMRES, for the solution of the linear systems. VODPK is a powerful tool for large stiff systems 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 [14]. The capabilities of both VODE and VODPK have been combined in the C-language package CVODE [21].

At present, CVODE 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) [50], FGMRES (Flexible Generalized Minimum RESidual) [49], Bi-CGStab (Bi-Conjugate Gradient Stabilized) [53], TFQMR (Transpose-Free Quasi-Minimal Residual) [29], and PCG (Preconditioned Conjugate Gradient) [31] linear iterative methods. As Krylov methods, these require almost no 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 stiff ODE systems, the Krylov methods are preferable over direct linear solver methods, and are often the only feasible choice. Among the Krylov methods in SUNDIALS, we recommend GMRES as the best overall choice. However, users are encouraged to compare all options, especially if encountering convergence failures with GMRES. Bi-CGStab and TFQMR have an advantage in storage requirements, in that the number of workspace vectors they require is fixed, while that number for GMRES depends on the desired Krylov subspace size. FGMRES has an advantage in that it is designed to support preconditioners that vary between iterations (e.g. iterative methods). PCG exhibits rapid convergence and minimal workspace vectors, but only works for symmetric linear systems.

In the process of translating the VODE and VODPK algorithms into C, the overall CVODE organization has been changed considerably. One key feature of the CVODE organization is that the linear system solvers comprise a layer of code modules that is separated from the integration algorithm, allowing for easy modification and expansion of the linear solver array. A second key feature is a separate module devoted to vector operations; this facilitated the extension to multiprosessor environments with minimal impacts on the rest of the solver, resulting in PVODE [19], the parallel variant of CVODE.

Around 2002, the functionality of CVODE and PVODE were combined into one single code, simply called CVODE. Development of this version of CVODE was concurrent with a redesign of the vector operations module across the SUNDIALS suite. The key feature of the NVECTOR module is that it is written in terms of abstract vector operations with the actual vector kernels attached by a particular implementation (such as serial or parallel) of NVECTOR. This allows writing the SUNDIALS solvers in a manner independent of the actual NVECTOR implementation (which can be user-supplied), as well as allowing more than one NVECTOR module linked into an executable file. SUNDIALS (and thus CVODE) is supplied with six different NVECTOR implementations: serial, MPI-parallel, and both OpenMP and Pthreads thread-parallel NVECTOR implementations, a Hypre parallel implementation, and a PETSc implementation.

There are several motivations for choosing the C language for CVODE. 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 CVODE 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.

1.2 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 SUNDIALSConfig.cmake file.

Added support for SuperLU DIST 6.3.0 or newer.

Changes in v5.4.0

Added new functions CVodeComputeState, and CVodeGetNonlinearSystemData which advanced users might find useful if providing a custom SUNNonlinSolSysFn.

Added the function CVodeSetLSNormFactor 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 CVodeGetNumNonlinSolvIters, the cumulative number of failures with CVodeGetNumNonlinSolvConvFails, or both with CVodeGetNonlinSolvStats.

A minor inconsistency in checking the Jacobian evaluation frequency has been fixed. As a result codes using using a non-default Jacobian update frequency through a call to CVodeSetMaxStepsBetweenJac will need to increase the provided value by 1 to achieve the same behavior as before. For greater clarity the function CVodeSetMaxStepsBetweenJac has been deprecated and replaced with CVodeSetJacEvalFrequency. Additionally, the function CVodeSetLSetupFrequency has been added to set the frequency of calls to the linear solver setup function.

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 11.

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 specialized fused CUDA kernels to CVODE which may offer better performance on smaller problems when using CVODE with the NVECTOR_CUDA module. See the optional input function CVodeSetUseIntegratorFusedKernels for more information. As with other SUNDIALS CUDA features, this capability is considered experimental and may change from version to version.

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 a new function, CVodeSetMonitorFn, that takes a user-function to be called by CVODE after every *nst* successfully completed time-steps. This is intended to provide a way of monitoring the CVODE statistics throughout the simulation.

Added a new function CVodeGetLinSolveStats to get the CVODE linear solver statistics as a group.

Added the optional function CVodeSetJacTimesRhsFn to specify an alternative right-hand side function for computing Jacobian-vector products with the internal difference quotient approximation.

Added support for integrating IVPs with constraints using BDF methods and projecting the solution onto the constraint manifold with a user defined projection function. This implementation is accompanied by additions to user documentation and CVODE examples. See §2.2 for more information.

Added support for CUDA v11.

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 CVodeSetLinearSolutionScaling 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 γ in the linear system matrix $I - \gamma J$. Scaling is enabled by default when using a matrix-based linear solver with BDF methods.

Changes in v5.1.0

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

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

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

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

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

Added support for constant damping to the SUNNonlinearSolver_FixedPoint module when using Anderson acceleration. See Section 10.4.1 and the SUNNonlinSolSetDamping_FixedPoint function for more details.

Changes in v5.0.0

Build system changes

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

• Fixed a bug in the build system that prevented the NVECTOR_PTHREADS module from being built.

NVECTOR module changes

- Two new functions were added to aid in creating custom NVECTOR objects. The constructor N_VNewEmpty allocates an "empty" generic NVECTOR with the object's content pointer and the function pointers in the operations structure initialized to NULL. When used in the constructor for custom objects this function will ease the introduction of any new optional operations to the NVECTOR API by ensuring only required operations need to be set. Additionally, the function N_VCopyOps(w, v) has been added to copy the operation function pointers between vector objects. When used in clone routines for custom vector objects these functions also will ease the introduction of any new optional operations to the NVECTOR API by ensuring all operations are copied when cloning objects. See §7.1.6 for more details.
- Two new NVECTOR implementations, NVECTOR_MANYVECTOR and NVECTOR_MPIMANYVECTOR, have been created to support flexible partitioning of solution data among different processing elements (e.g., CPU + GPU) or for multi-physics problems that couple distinct MPI-based simulations together. This implementation is accompanied by additions to user documentation and SUNDIALS examples. See §7.15 and §7.16 for more details.
- One new required vector operation and ten new optional vector operations have been added to the NVECTOR API. The new required operation, N_VGetLength, returns the global length of an N_Vector. The optional operations have been added to support the new NVECTOR_MPIMANYVECTOR implementation. The operation N_VGetCommunicator must be implemented by subvectors that are combined to create an NVECTOR_MPIMANYVECTOR, but is not used outside of this context. The remaining nine operations are optional local reduction operations intended to eliminate unnecessary latency when performing vector reduction operations (norms, etc.) on distributed memory systems. The optional local reduction vector operations are N_VDotProdLocal, N_VMaxNormLocal, N_VMinLocal, N_VL1NormLocal, N_VWSqrSumLocal, N_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 §10.5 for more details.
- Added new Fortran 2003 interfaces for most Sunnonlinsol modules. See Chapter 10 for more details on how to use the interfaces.

CVODE changes

- Fixed a bug in the CVODE constraint handling where the step size could be set below the minimum step size.
- Fixed a bug in the CVODE nonlinear solver interface where the norm of the accumulated correction was not updated when using a non-default convergence test function.
- Fixed a memeory leak in FCVODE 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, CVODE will remove at least one global reduction per time step.
- The CVLS 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.
- A new linear solver interface function CVLsLinSysFn was added as an alternative method for evaluating the linear system $M = I \gamma J$.

• Added two new functions, CVodeGetCurrentGamma and CVodeGetCurrentState, which may be useful to users who choose to provide their own nonlinear solver implementations.

• The CVODE Fortran 2003 interface was completely redone to be more sustainable and to allow users to write more idiomatic Fortran. 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 cvode_impl.h is no longer installed. This means users who are directly manipulating the CVodeMem structure will need to update their code to use CVODE'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 CVODE library, libsundials_cvode.

Changes in v4.0.1

No changes were made in this release.

Changes in v4.0.0

CVODE's previous direct and iterative linear solver interfaces, CVDLS and CVSPILS, have been merged into a single unified linear solver interface, CVLS, to support any valid SUNLINSOL module. This includes the "DIRECT" and "ITERATIVE" types as well as the new "MATRIX_ITERATIVE" type. Details regarding how CVLS 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 CVODE example programs and the standalone linear solver examples have been updated to use the unified linear solver interface.

The unified interface for the new CVLS module is very similar to the previous CVDLS and CVSPILS interfaces. To minimize challenges in user migration to the new names, the previous C and FORTRAN routine names may still be used; these will be deprecated in future releases, so we recommend that users migrate to the new names soon. Additionally, we note that FORTRAN users, however, may need to enlarge their iout array of optional integer outputs, and update the indices that they query for certain linear-solver-related statistics.

The names of all constructor routines for SUNDIALS-provided SUNLINSOL implementations have been updated to follow the naming convention SUNLinSol_* where * is the name of the linear solver. The new names are SUNLinSol_Band, SUNLinSol_Dense, SUNLinSol_KLU, SUNLinSol_LapackBand, SUNLinSol_LapackDense, SUNLinSol_PCG, SUNLinSol_SPBCGS, SUNLinSol_SPFGMR, SUNLinSol_SPGMR, SUNLinSol_SPFGMR, 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 CVODE 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 CVODE example programs have been updated to use generic SUNNONLINSOL modules.

With the introduction of SUNNONLINSOL modules, the input parameter iter to CVodeCreate has been removed along with the function CVodeSetIterType and the constants CV_NEWTON and CV_FUNCTIONAL. Similarly, the ITMETH parameter has been removed from the Fortran interface function FCVMALLOC. Instead of specifying the nonlinear iteration type when creating the CVODE memory structure, CVODE uses the SUNNONLINSOL_NEWTON module implementation of a Newton iteration by default. For details on using a non-default or user-supplied nonlinear solver see Chapter 4. CVODE functions for setting the nonlinear solver options (e.g., CVodeSetMaxNonlinIters) or getting nonlinear solver statistics (e.g., CVodeGetNumNonlinSolvIters) remain unchanged and internally call generic SUNNONLINSOL functions as needed.

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.

Two changes were made in the CVODE/CVODES/ARKODE initial step size algorithm:

- 1. Fixed an efficiency bug where an extra call to the right hand side function was made.
- 2. Changed the behavior of the algorithm if the max-iterations case is hit. Before the algorithm would exit with the step size calculated on the penultimate iteration. Now it will exit with the step size calculated on the final iteration.

A FORTRAN 2003 interface to CVODE has been added along with FORTRAN 2003 interfaces to the following shared SUNDIALS modules:

- SUNNONLINSOL_FIXEDPOINT and SUNNONLINSOL_NEWTON nonlinear solver modules
- SUNLINSOL_DENSE, SUNLINSOL_BAND, SUNLINSOL_KLU, SUNLINSOL_PCG, SUNLINSOL_SPBCGS, SUNLINSOL_SPFGMR, SUNLINSOL_SPGMR, and SUNLINSOL_SPTFQMR linear solver modules
- NVECTOR_SERIAL, NVECTOR_PTHREADS, and NVECTOR_OPENMP vector modules

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

Support for optional inequality constraints on individual components of the solution vector has been added to CVODE and CVODES. See Chapter 2 and the description of CVodeSetConstraints in §4.5.8.1 for more details. Use of CVodeSetConstraints requires the NVECTOR operations N_MinQuotient, N_VConstrMask, and N_VCompare that were not previously required by CVODE and CVODES.

Fixed a problem with setting sunindextype which would occur with some compilers (e.g. arm-clang) 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 "ScaleAddI" implementations in the sparse SUNMatrix module to more optimally handle the case where the target matrix contained sufficient storage for the sum, but had the wrong sparsity pattern. The sum now occurs in-place, by performing the sum backwards in the existing storage. However, it is still more efficient if the user-supplied Jacobian routine allocates storage for the sum $I + \gamma J$ manually (with zero entries if needed).
- Added the following examples from the usage notes page of the SUNDIALS website, and updated them to work with SUNDIALS 3.x:
 - cvDisc_dns.c, which demonstrates using CVODE with discontinuous solutions or RHS.
 - cvRoberts_dns_negsol.c, which illustrates the use of the RHS function return value to control unphysical negative concentrations.
- 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 minor bug in the cvSLdet routine, where a return was missing in the error check for three inconsistent roots.

- 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.
- Fixed an indexing bug in the CUDA NVECTOR implementation of N_VWrmsNormMask and revised the RAJA NVECTOR implementation of N_VWrmsNormMask to work with mask arrays using values other than zero or one. Replaced double with realtype in the RAJA vector test functions.
- Fixed compilation issue with GCC 7.3.0 and Fortran programs that do not require a SUNMATRIX or SUNLINSOL module (e.g., iterative linear solvers or fixed-point iteration).

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

Changes in v3.1.0

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

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

Changes in v3.0.0

All interfaces to matrix structures and linear solvers have been reworked, and all example programs have been updated. The goal of the redesign of these interfaces was to provide more encapsulation and ease in interfacing 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: dense, banded, LAPACK dense, LAPACK band, KLU, SuperLU_MT, SPGMR, SPBCGS, SPTFQMR, SPFGMR, PCG. These replicate previous SUNDIALS generic linear solvers in a single objectoriented API.
- Added example problems demonstrating use of generic SUNLINEARSOLVER modules.
- Expanded package-provided direct linear solver (Dls) interfaces and scaled, preconditioned, iterative linear solver (Spils) interfaces to utilize generic SUNMATRIX and SUNLINEARSOLVER objects.

- Removed package-specific, linear solver-specific, solver modules (e.g. CVDENSE, KINBAND, IDAKLU, ARKSPGMR) since their functionality is entirely replicated by the generic Dls/Spils interfaces and SUNLINEARSOLVER/SUNMATRIX modules. The exception is CVDIAG, a diagonal approximate Jacobian solver available to CVODE and CVODES.
- Converted all SUNDIALS example problems to utilize new generic SUNMATRIX and SUNLIN-EARSOLVER 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 th web site, https://software.llnl.gov/RAJA/. These additions are accompanied by additions to various interface functions and to user documentation.

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

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

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

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

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

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.

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

A bug fix was made in CVodeFree to call 1free unconditionally (if non-NULL).

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.

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

In FCVODE, corrections were made to three Fortran interface functions. Missing Fortran interface routines were added so that users can supply the sparse Jacobian routine when using sparse direct solvers

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 and for use of sparse direct solvers from Fortran.

Minor corrections and additions were made to the CVODE 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 CVODE 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 CVODE.

Otherwise, only relatively minor modifications were made to the CVODE solver:

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

In order to eliminate or minimize the differences between the sources for private functions in CVODE and CVODES, the names of 48 private functions were changed from CV** to cv**, and a few other names were also changed.

Two minor bugs were fixed regarding the testing of input on the first call to CVode – one involving tstop and one involving the initialization of *tret.

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.

The example program cvAdvDiff_diag_p was added to illustrate the use of CVDiag in parallel.

In the FCVODE optional input routines FCVSETIIN and FCVSETRIN, the optional fourth argument key_length was removed, with hardcoded key string lengths passed to all strncmp tests.

In all FCVODE 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: In CVSetTqBDF, the logic was changed to avoid a divide by zero. After the solver memory is created, it is set to zero before being filled. In each linear solver interface function, the linear solver memory is freed on an error return, and the **Free function now includes a line setting to NULL the main memory pointer to the linear solver memory. In the rootfinding functions CVRcheck1/CVRcheck2, when an exact zero is found, the array glo of g values at the left endpoint is adjusted, instead of shifting the t location tlo slightly. In the installation files, we modified the treatment of the macro SUNDIALS_USE_GENERIC_MATH, so that the parameter GENERIC_MATH_LIB is either defined (with no value) or not defined.

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) an 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 existing 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; and (c) a general streamlining of the preconditioner modules 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 instaltion *include* directory.

The functions in the generic dense linear solver (sundials_dense and sundials_smalldense) were modified to work for rectangular $m \times n$ matrices ($m \le n$), 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

CVSPBCG and CVSPTFQMR 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). Corresponding additions were made to the FORTRAN interface module FCVODE. 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 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 (cvode_ 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 v2.2.1

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 sundless suite. We have eliminated the mechanism of providing optional inputs and extracting optional statistics from the solver through the iopt and ropt arrays. Instead, CVODE now provides a set of routines (with prefix CVodeSet) to change the default values for various quantities controlling the solver and a set of extraction routines (with prefix CVodeGet) 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.

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

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

1.3 Reading this User Guide

This user guide is a combination of general usage instructions. Specific example programs are provided as a separate document. We expect that some readers will want to concentrate on the general instructions, while others will refer mostly to the examples, and the organization is intended to accommodate both styles.

There are different possible levels of usage of CVODE. The most casual user, with a small IVP problem only, can get by with reading §2.1, then Chapter 4 through §4.5.7 only, and looking at examples in [38].

In a different direction, a more expert user with an IVP problem may want to (a) use a package preconditioner (§4.7), (b) supply his/her own Jacobian or preconditioner routines (§4.6), (c) do multiple runs of problems of the same size (§4.5.11), (d) supply a new NVECTOR module (Chapter 7), (e) supply new SUNLINSOL and/or SUNMATRIX modules (Chapters 8 and 9), or even (f) supply new SUNNONLINSOL modules (Chapter 10).

The structure of this document is as follows:

- In Chapter 2, we give short descriptions of the numerical methods implemented by CVODE for the solution of initial value problems for systems of ODEs, and continue with short descriptions of preconditioning (§2.3), stability limit detection (§2.4), and rootfinding (§2.5).
- The following chapter describes the structure of the SUNDIALS suite of solvers (§3.1) and the software organization of the CVODE solver (§3.2).
- Chapter 4 is the main usage document for CVODE for C applications. It includes a complete description of the user interface for the integration of ODE initial value problems.
- In Chapter 5, we describe the use of CVODE with FORTRAN applications.

- Chapter 7 gives a brief overview of the generic NVECTOR module shared among the various components of SUNDIALS, and details on the 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 CVODE, within the structure of SUNDIALS (Appendix A), as well as a list of all the constants used for input to and output from CVODE functions (Appendix B).

Finally, the reader should be aware of the following notational conventions in this user guide: program listings and identifiers (such as CVodeInit) within textual explanations appear in typewriter type style; fields in C structures (such as content) appear in italics; and packages or modules, such as CVLS, 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 CVODE and PVODE codes and their user guides by Scott D. Cohen [20] and George D. Byrne [18].

1.4 SUNDIALS Release License

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

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



1.4.1 BSD 3-Clause License

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

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

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

* Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

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

1.4.2 Additional Notice

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

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

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

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

1.4.3 SUNDIALS Release Numbers

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

Chapter 2

Mathematical Considerations

CVODE solves ODE initial value problems (IVPs) in real N-space, which we write in the abstract form

$$\dot{y} = f(t, y), \quad y(t_0) = y_0,$$
 (2.1)

where $y \in \mathbf{R}^N$ and $f: \mathbf{R} \times \mathbf{R}^N \to \mathbf{R}^N$. Here we use \dot{y} to denote dy/dt. While we use t to denote the independent variable, and usually this is time, it certainly need not be. CVODE solves both stiff and nonstiff systems. Roughly speaking, stiffness is characterized by the presence of at least one rapidly damped mode, whose time constant is small compared to the time scale of the solution itself. Additionally, CVODE solves ODE IVPs with constraints in the explicit form

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

 $0 = g(t, y)$ (2.2)

where $g(t,y): \mathbf{R} \times \mathbf{R}^N \to \mathbf{R}^M$ and the initial condition lies on the constraint manifold i.e., $g(t_0,y_0) = 0$.

2.1 IVP solution

The methods used in CVODE are variable-order, variable-step multistep methods, based on formulas of the form

$$\sum_{i=0}^{K_1} \alpha_{n,i} y^{n-i} + h_n \sum_{i=0}^{K_2} \beta_{n,i} \dot{y}^{n-i} = 0.$$
 (2.3)

Here the y^n are computed approximations to $y(t_n)$, and $h_n = t_n - t_{n-1}$ is the step size. The user of CVODE must choose appropriately one of two multistep methods. For nonstiff problems, CVODE includes the Adams-Moulton formulas, characterized by $K_1 = 1$ and $K_2 = q - 1$ above, where the order q varies between 1 and 12. For stiff problems, CVODE includes the Backward Differentiation Formulas (BDF) in so-called fixed-leading coefficient (FLC) form, given by $K_1 = q$ and $K_2 = 0$, with order q varying between 1 and 5. The coefficients are uniquely determined by the method type, its order, the recent history of the step sizes, and the normalization $\alpha_{n,0} = -1$. See [17] and [40].

For either choice of formula, a nonlinear system must be solved (approximately) at each integration step. This nonlinear system can be formulated as either a rootfinding problem

$$F(y^n) \equiv y^n - h_n \beta_{n,0} f(t_n, y^n) - a_n = 0, \qquad (2.4)$$

or as a fixed-point problem

$$G(y^n) \equiv h_n \beta_{n,0} f(t_n, y^n) + a_n = y^n$$
. (2.5)

where $a_n \equiv \sum_{i>0} (\alpha_{n,i} y^{n-i} + h_n \beta_{n,i} \dot{y}^{n-i})$. CVODE provides several nonlinear solver choices as well as the option of using a user-defined nonlinear solver (see Chapter 10). By default CVODE solves (2.4) with a *Newton iteration* which requires the solution of linear systems

$$M[y^{n(m+1)} - y^{n(m)}] = -F(y^{n(m)}), (2.6)$$

in which

$$M \approx I - \gamma J$$
, $J = \partial f / \partial y$, and $\gamma = h_n \beta_{n,0}$. (2.7)

The exact variation of the Newton iteration depends on the choice of linear solver and is discussed below and in §10.3. For nonstiff systems, a fixed-point iteration (previously referred to as a functional iteration in this guide) solving (2.5) is also available. This involves evaluations of f only and can (optionally) use Anderson's method [10, 54, 28, 46] to accelerate convergence (see §10.4 for more details). For any nonlinear solver, the initial guess for the iteration is a predicted value $y^{n(0)}$ computed explicitly from the available history data.

For nonlinear solvers that require the solution of the linear system (2.6) (e.g., the default Newton iteration), CVODE provides several linear solver 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 [43, 24, 9] (serial or threaded vector modules only) [Note that users will need to download and install the KLU or SUPERLUMT packages independent of CVODE],
- Spgmr, a scaled preconditioned GMRES (Generalized Minimal Residual method) solver,
- SPFGMR, a scaled preconditioned FGMRES (Flexible Generalized Minimal Residual method) solver,
- SPBCGS, a scaled preconditioned Bi-CGStab (Bi-Conjugate Gradient Stable method) solver,
- SPTFQMR, a scaled preconditioned TFQMR (Transpose-Free Quasi-Minimal Residual method) solver, or
- PCG, a scaled preconditioned CG (Conjugate Gradient method) solver.

For large stiff systems, where direct methods are often 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 [14].

In addition, CVODE also provides a linear solver module which only uses a diagonal approximation of the Jacobian matrix.

Note that the dense, band, and sparse direct linear solvers can only be used with the serial and threaded vector representations. The diagonal solver can be used with any vector representation.

In the process of controlling errors at various levels, CVODE 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

$$W_i = 1/[\text{RTOL} \cdot |y_i| + \text{ATOL}_i]. \tag{2.8}$$

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 iteration matrix M is fixed throughout the nonlinear iterations. However, in

2.1 IVP solution 21

the case that a matrix-free iterative linear solver is used, the default Newton iteration is an Inexact Newton iteration, in which M is applied in a matrix-free manner, with matrix-vector products Jv obtained by either difference quotients or a user-supplied routine. With the default Newton iteration, the matrix M 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,
- more than 20 steps have been taken since the last update,
- the value $\bar{\gamma}$ of γ at the last update satisfies $|\gamma/\bar{\gamma}-1|>0.3$,
- a non-fatal convergence failure just occurred, or
- an error test failure just occurred.

When forced by a convergence failure, an update of M or P may or may not involve a reevaluation of J (in M) or of Jacobian data (in P), depending on whether Jacobian error was the likely cause of the failure. More generally, the decision is made to reevaluate J (or instruct the user to reevaluate Jacobian data in P) when:

- starting the problem,
- more than 50 steps have been taken since the last evaluation,
- a convergence failure occurred with an outdated matrix, and the value $\bar{\gamma}$ of γ at the last update satisfies $|\gamma/\bar{\gamma}-1| < 0.2$, or
- a convergence failure occurred that forced a step size reduction.

The default stopping test for nonlinear solver iterations is related to the subsequent local error test, with the goal of keeping the nonlinear iteration errors from interfering with local error control. As described below, the final computed value $y^{n(m)}$ will have to satisfy a local error test $||y^{n(m)} - y^{n(0)}|| \le \epsilon$. Letting y^n denote the exact solution of (2.4), we want to ensure that the iteration error $y^n - y^{n(m)}$ is small relative to ϵ , specifically that it is less than 0.1ϵ . (The safety factor 0.1 can be changed by the user.) For this, we also estimate the linear convergence rate constant R as follows. We initialize R to 1, and reset R = 1 when M or P is updated. After computing a correction $\delta_m = y^{n(m)} - y^{n(m-1)}$, we update R if m > 1 as

$$R \leftarrow \max\{0.3R, \|\delta_m\|/\|\delta_{m-1}\|\}$$
.

Now we use the estimate

$$||y^n - y^{n(m)}|| \approx ||y^{n(m+1)} - y^{n(m)}|| \approx R||y^{n(m)} - y^{n(m-1)}|| = R||\delta_m||.$$

Therefore the convergence (stopping) test is

$$R\|\delta_m\| < 0.1\epsilon$$
.

We allow at most 3 iterations (but this limit can be changed by the user). We also declare the iteration diverged if any $\|\delta_m\|/\|\delta_{m-1}\| > 2$ with m > 1. If convergence fails with J or P current, we are forced to reduce the step size, and we replace h_n by $h_n/4$. The integration is halted after a preset number of convergence failures; the default value of this limit is 10, but this can be changed by the user.

When an iterative method is used to solve the linear system, its errors must also be controlled, and this also involves the local error test constant. The linear iteration error in the solution vector δ_m is approximated by the preconditioned residual vector. Thus to ensure (or attempt to ensure) that the linear iteration errors do not interfere with the nonlinear error and local integration error controls, we require that the norm of the preconditioned residual be less than $0.05 \cdot (0.1\epsilon)$.

When the Jacobian is stored using either dense or band SUNMATRIX objects, the Jacobian may be supplied by a user routine, or approximated by difference quotients, at the user's option. In the latter case, we use the usual approximation

$$J_{ij} = [f_i(t, y + \sigma_j e_j) - f_i(t, y)]/\sigma_j.$$

The increments σ_i are given by

$$\sigma_j = \max \left\{ \sqrt{U} |y_j|, \sigma_0/W_j \right\},$$

where U is the unit roundoff, σ_0 is a dimensionless value, and W_j is the error weight defined in (2.8). In the dense case, this scheme requires N evaluations of f, one for each column of J. In the band case, the columns of J are computed in groups, by the Curtis-Powell-Reid algorithm, with the number of f evaluations equal to the bandwidth.

We note that with sparse and user-supplied SUNMATRIX objects, the Jacobian *must* be supplied by a user routine.

In the case of a Krylov method, preconditioning may be used on the left, on the right, or both, with user-supplied routines for the preconditioning setup and solve operations, and optionally also for the required matrix-vector products Jv. If a routine for Jv is not supplied, these products are computed as

$$Jv = [f(t, y + \sigma v) - f(t, y)]/\sigma.$$
(2.9)

The increment σ is $1/\|v\|$, so that σv has norm 1.

A critical part of CVODE — making it an ODE "solver" rather than just an ODE method, is its control of local error. At every step, the local error is estimated and required to satisfy tolerance conditions, and the step is redone with reduced step size whenever that error test fails. As with any linear multistep method, the local truncation error LTE, at order q and step size h, satisfies an asymptotic relation

$$LTE = Ch^{q+1}y^{(q+1)} + O(h^{q+2})$$

for some constant C, under mild assumptions on the step sizes. A similar relation holds for the error in the predictor $y^{n(0)}$. These are combined to get a relation

LTE =
$$C'[y^n - y^{n(0)}] + O(h^{q+2})$$
.

The local error test is simply $\|\text{LTE}\| \le 1$. Using the above, it is performed on the predictor-corrector difference $\Delta_n \equiv y^{n(m)} - y^{n(0)}$ (with $y^{n(m)}$ the final iterate computed), and takes the form

$$\|\Delta_n\| \le \epsilon \equiv 1/|C'|.$$

If this test passes, the step is considered successful. If it fails, the step is rejected and a new step size h' is computed based on the asymptotic behavior of the local error, namely by the equation

$$(h'/h)^{q+1} \|\Delta_n\| = \epsilon/6.$$

Here 1/6 is a safety factor. A new attempt at the step is made, and the error test repeated. If it fails three times, the order q is reset to 1 (if q > 1), or the step is restarted from scratch (if q = 1). The ratio h'/h is limited above to 0.2 after two error test failures, and limited below to 0.1 after three. After seven failures, CVODE returns to the user with a give-up message.

In addition to adjusting the step size to meet the local error test, CVODE periodically adjusts the order, with the goal of maximizing the step size. The integration starts out at order 1 and varies the order dynamically after that. The basic idea is to pick the order q for which a polynomial of order q best fits the discrete data involved in the multistep method. However, if either a convergence failure or an error test failure occurred on the step just completed, no change in step size or order is done. At the current order q, selecting a new step size is done exactly as when the error test fails, giving a tentative step size ratio

$$h'/h = (\epsilon/6||\Delta_n||)^{1/(q+1)} \equiv \eta_q$$
.

We consider changing order only after taking q+1 steps at order q, and then we consider only orders q'=q-1 (if q>1) or q'=q+1 (if q<5). The local truncation error at order q' is estimated using the history data. Then a tentative step size ratio is computed on the basis that this error, LTE(q'), behaves asymptotically as $h^{q'+1}$. With safety factors of 1/6 and 1/10 respectively, these ratios are:

$$h'/h = [1/6||\text{LTE}(q-1)||]^{1/q} \equiv \eta_{q-1}$$

and

$$h'/h = [1/10||\text{LTE}(q+1)||]^{1/(q+2)} \equiv \eta_{q+1}$$
.

The new order and step size are then set according to

$$\eta = \max\{\eta_{q-1}, \eta_q, \eta_{q+1}\}, \quad h' = \eta h,$$

with q' set to the index achieving the above maximum. However, if we find that $\eta < 1.5$, we do not bother with the change. Also, h'/h is always limited to 10, except on the first step, when it is limited to 10^4 .

The various algorithmic features of CVODE described above, as inherited from VODE and VODPK, are documented in [13, 16, 35]. They are also summarized in [36].

CVODE 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: $y_i > 0$, $y_i < 0$, $y_i \ge 0$, or $y_i \le 0$. The constraint satisfaction is tested after a successful nonlinear system solution. If any constraint fails, we declare a convergence failure of the Newton iteration and reduce the step size. Rather than cutting the step size by some arbitrary factor, CVODE 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). If a step fails to satisfy the constraints repeatedly within a step attempt or fails with the minimum step size 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, CVODE takes steps until a user-defined output value $t = t_{\text{out}}$ is overtaken, and then it 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 CVODE not to integrate past a given stopping point $t = t_{\text{stop}}$.

2.2 IVPs with constraints

For IVPs with constraints CVODE ensures that the solution satisfies the constraint equation by projecting a successfully computed time step onto the invariant manifold. As discussed in [27] and [51], this approach reduces the error in the solution and retains the order of convergence of the numerical method. Therefore, in an attempt to advance the solution to a new point in time (i.e., taking a new integration step), CVODE performs the following operations:

- 1. predict solution
- 2. solve nonlinear system and correct solution
- 3. project solution
- 4. test error
- 5. select order and step size for next step

and includes several recovery attempts in case there are convergence failures (or difficulties) in the nonlinear solver or in the projection step, or if the solution fails to satisfy the error test. Note that at this time projection is only supported with BDF methods and the projection function must be user-defined. See §4.5.6 and §4.6.6 for more information on providing a projection function to CVODE.

When using a coordinate projection method the solution y_n is obtained by projecting (orthogonally or otherwise) the solution \tilde{y}_n from step 2 above onto the manifold given by the constraint. As such y_n is computed as the solution of the nonlinear constrained least squares problem

minimize
$$||y_n - \tilde{y}_n||$$

subject to $g(t_n, y_n) = 0$. (2.10)

The solution of (2.10) can be computed iteratively with a Newton method. Given an initial guess $y_n^{(0)}$ the iterations are computed as

$$y_n^{(i+1)} = y_n^{(i)} + \delta y_n^{(i)} \tag{2.11}$$

where the increment $\delta y_n^{(i)}$ is the solution of the least-norm problem

minimize
$$\|\delta y_n^{(i)}\|$$

subject to $G(t_n, y_n^{(i)}) \delta y_n^{(i)} = -g(t_n, y_n^{(i)})$ (2.12)

where $G(t, y) = \partial g(t, y)/\partial y$.

If the projected solution satisfies the error test then the step is accepted and the correction to the unprojected solution, $\Delta_p = y_n - \tilde{y}_n$, is used to update the Nordsieck history array for the next step.

2.3 Preconditioning

When using a nonlinear solver that requires the solution of the linear system (2.6) (e.g., the default Newton iteration), CVODE makes repeated use of a linear solver to solve linear systems of the form Mx = -r, where x is a correction vector and r is a residual vector. 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, as $(P^{-1}A)x = P^{-1}b$; on the right, as $(AP^{-1})Px = b$; or on both sides, as $(P_L^{-1}AP_R^{-1})P_Rx = P_L^{-1}b$. 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 P_LP_R in the last case, should in some sense approximate the system matrix P_L or the same time, in order to be cost-effective, the matrix P_L or matrices P_L and P_R , 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 [14] for an extensive study of preconditioners for reaction-transport systems).

Most of the iterative linear solvers supplied with SUNDIALS allow for preconditioning either side, or on both sides, although we know of no situation where preconditioning on both sides is clearly superior to preconditioning on one side only (with the product P_LP_R). Moreover, for a given preconditioner matrix, the merits of left vs. right preconditioning are unclear in general, and the user should experiment with both choices. Performance will differ because the inverse of the left preconditioner is included in the linear system residual whose norm is being tested in the Krylov algorithm. As a rule, however, if the preconditioner is the product of two matrices, we recommend that preconditioning be done either on the left only or the right only, rather than using one factor on each side.

Typical preconditioners used with CVODE are based on approximations to the system Jacobian, $J = \partial f/\partial y$. Since the matrix involved is $M = I - \gamma J$, any approximation \bar{J} to J yields a matrix that is of potential use as a preconditioner, namely $P = I - \gamma \bar{J}$. 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.

2.4 BDF stability limit detection

CVODE includes an algorithm, STALD (STAbility Limit Detection), which provides protection against potentially unstable behavior of the BDF multistep integration methods in certain situations, as described below.

When the BDF option is selected, CVODES uses Backward Differentiation Formula methods of orders 1 to 5. At order 1 or 2, the BDF method is A-stable, meaning that for any complex constant λ in the open left half-plane, the method is unconditionally stable (for any step size) for the standard scalar model problem $\dot{y} = \lambda y$. For an ODE system, this means that, roughly speaking, as long as all modes in the system are stable, the method is also stable for any choice of step size, at least in the sense of a local linear stability analysis.

At orders 3 to 5, the BDF methods are not A-stable, although they are *stiffly stable*. In each case, in order for the method to be stable at step size h on the scalar model problem, the product $h\lambda$ must lie within a region of absolute stability. That region excludes a portion of the left half-plane that is concentrated near the imaginary axis. The size of that region of instability grows as the order increases from 3 to 5. What this means is that, when running BDF at any of these orders, if an eigenvalue λ of the system lies close enough to the imaginary axis, the step sizes h for which the method is stable are limited (at least according to the linear stability theory) to a set that prevents $h\lambda$ from leaving the stability region. The meaning of close enough depends on the order. At order 3, the unstable region is much narrower than at order 5, so the potential for unstable behavior grows with order.

System eigenvalues that are likely to run into this instability are ones that correspond to weakly damped oscillations. A pure undamped oscillation corresponds to an eigenvalue on the imaginary axis. Problems with modes of that kind call for different considerations, since the oscillation generally must be followed by the solver, and this requires step sizes ($h \sim 1/\nu$, where ν is the frequency) that are stable for BDF anyway. But for a weakly damped oscillatory mode, the oscillation in the solution is eventually damped to the noise level, and at that time it is important that the solver not be restricted to step sizes on the order of $1/\nu$. It is in this situation that the new option may be of great value.

In terms of partial differential equations, the typical problems for which the stability limit detection option is appropriate are ODE systems resulting from semi-discretized PDEs (i.e., PDEs discretized in space) with advection and diffusion, but with advection dominating over diffusion. Diffusion alone produces pure decay modes, while advection tends to produce undamped oscillatory modes. A mix of the two with advection dominant will have weakly damped oscillatory modes.

The STALD algorithm attempts to detect, in a direct manner, the presence of a stability region boundary that is limiting the step sizes in the presence of a weakly damped oscillation [33]. The algorithm supplements (but differs greatly from) the existing algorithms in CVODES for choosing step size and order based on estimated local truncation errors. The STALD algorithm works directly with history data that is readily available in CVODE. If it concludes that the step size is in fact stability-limited, it dictates a reduction in the method order, regardless of the outcome of the error-based algorithm. The STALD algorithm has been tested in combination with the VODE solver on linear advection-dominated advection-diffusion problems [34], where it works well. The implementation in CVODE has been successfully tested on linear and nonlinear advection-diffusion problems, among others.

This stability limit detection option adds some computational overhead to the CVODES solution. (In timing tests, these overhead costs have ranged from 2% to 7% of the total, depending on the size and complexity of the problem, with lower relative costs for larger problems.) Therefore, it should be activated only when there is reasonable expectation of modes in the user's system for which it is appropriate. In particular, if a CVODE solution with this option turned off appears to take an inordinately large number of steps at orders 3-5 for no apparent reason in terms of the solution time scale, then there is a good chance that step sizes are being limited by stability, and that turning on the option will improve the efficiency of the solution.

2.5 Rootfinding

The CVODE solver has been augmented to include a rootfinding feature. This means that, while integrating the Initial Value Problem (2.1), CVODE can also find the roots of a set of user-defined functions $g_i(t, y)$ that depend both on t and on the solution vector y = 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 $g_i(t, y(t))$, 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 CVODE. 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 hone in on the root(s) with a modified secant method [32]. In addition, each time g is computed, CVODE 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, CVODE computes g at $t + \delta$ for a small increment δ , slightly further in the direction of integration, and if any $g_i(t + \delta) = 0$ also, CVODE stops and reports an error. This way, each time CVODE 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, CVODE 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} , 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_i at t_{hi} for zeros and for sign changes in (t_{lo}, t_{hi}) . If no sign changes were 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 = 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 include the sign change. If there is none in (t_{lo}, t_{mid}) but some $g_i(t_{mid}) = 0$, then that root is reported. The loop continues until $|t_{hi} - t_{lo}| < \tau$, and then the reported root location is t_{hi} .

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

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

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

28 Code Organization

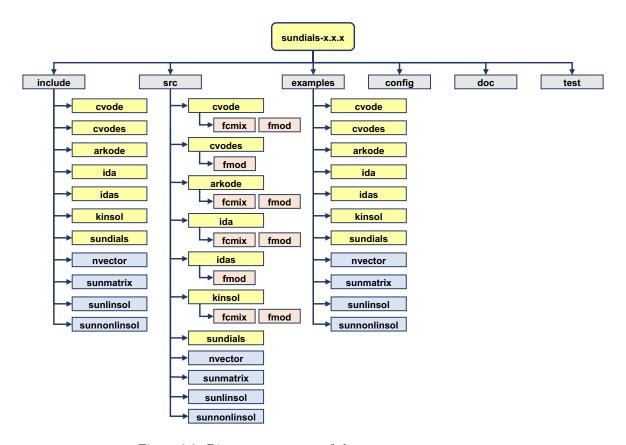


Figure 3.2: Directory structure of the SUNDIALS source tree.

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 CVODE organization

The CVODE package is written in ANSI C. The following summarizes the basic structure of the package, although knowledge of this structure is not necessary for its use.

The overall organization of the CVODE package is shown in Figure 3.3. The central integration module, implemented in the files cvode.h, cvode_impl.h, and cvode.c, deals with the evaluation of

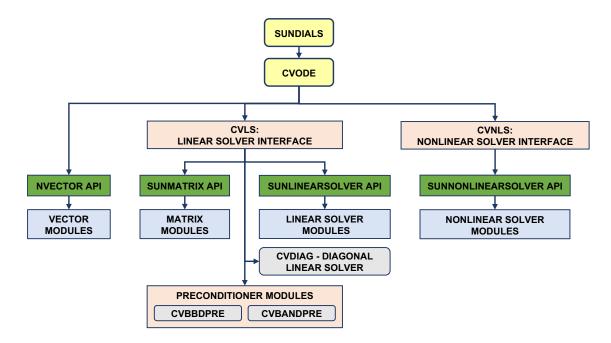


Figure 3.3: Overall structure diagram of the CVODE package. Modules specific to CVODE begin with "CV" (CVLS, CVNLS, CVDIAG, CVBBDPRE, and CVBANDPRE), all other items correspond to generic SUNDIALS vector, matrix, and solver modules (see Figure 3.1).

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

CVODE utilizes generic linear and nonlinear solver modules defined by the SUNLINSOL API (see Chapter 9) and SUNNONLINSOL API (see Chapter 10) respectively. As such, CVODE has no knowledge of the method being used to solve the linear and nonlinear systems that arise. For any given user problem, there exists a single nonlinear solver interface and, if necessary, one of the linear system solver interfaces is specified, and invoked as needed during the integration.

At present, the package includes two linear solver interfaces. The primary linear solver interface, CVLS, supports 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 CVODE can operate on any valid SUNLINSOL implementation, the set of linear solver modules available to CVODE will expand as new SUNLINSOL modules are developed.

Additionally, CVODE includes the *diagonal* linear solver interface, CVDIAG, that creates an internally generated diagonal approximation to the Jacobian.

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

For users employing matrix-free iterative linear solvers, CVODE includes an algorithm for the approximation by difference quotients of the product Mv. 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 [14, 16], together with the example and demonstration programs included with CVODE, offer considerable assistance in building preconditioners.

CVODE's linear solver interface consists of four primary phases, devoted to (1) memory allocation

30 Code Organization

and initialization, (2) setup of the matrix data involved, (3) solution of the system, and (4) freeing of memory. The setup and solution phases are separate because the evaluation of Jacobians and preconditioners is done only periodically during the integration, and only as required to achieve convergence.

CVODE also provides two preconditioner modules, for use with any of the Krylov iterative linear solvers. The first one, CVBANDPRE, is intended to be used with NVECTOR_SERIAL, NVECTOR_OPENMP or NVECTOR_PTHREADS and provides a banded difference-quotient Jacobian-based preconditioner, with corresponding setup and solve routines. The second preconditioner module, CVBBDPRE, 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 CVODE 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 CVODE 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 CVODE memory structure. The reentrancy of CVODE was motivated by the anticipated multicomputer extension, but is also essential in a uniprocessor setting where two or more problems are solved by intermixed calls to the package from within a single user program.

Chapter 4

Using CVODE for C Applications

This chapter is concerned with the use of CVODE for the solution of initial value problems (IVPs) in a C language setting. The following sections treat the header files and the layout of the user's main program, and provide descriptions of the CVODE user-callable functions and user-supplied functions.

The sample programs described in the companion document [38] may also be helpful. Those codes may be used as templates (with the removal of some lines used in testing) and are included in the CVODE package.

Users with applications written in FORTRAN should see Chapter 5, which describes interfacing with CVODE from FORTRAN.

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 CVBAND-PRE preconditioning module is only compatible with the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector implementations, and the preconditioner module CVBBDPRE 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.

CVODE 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 CVODE, 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 CVODE. The relevant library files are

- libdir/libsundials_cvode.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/cvode
- 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).

4.2 Data Types

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

4.2.1 Floating point types

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

Additionally, based on the current precision, sundials_types.h defines BIG_REAL to be the largest value representable as a realtype, SMALL_REAL to be the smallest value representable as a realtype, and UNIT_ROUNDOFF to be the difference between 1.0 and the minimum realtype greater than 1.0.

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

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

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

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

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

4.2.2 Integer types used for indexing

The type sunindextype is used for indexing array entries in SUNDIALS modules (e.g., vectors lengths and matrix sizes) as well as for storing the total problem size. During configuration sunindextype

4.3 Header files 33

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:

• cvode/cvode.h, the main header file for CVODE, which defines the several types and various constants, and includes function prototypes. This includes the header file for CVLS, cvode/cvode_ls.h.

Note that cvode.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 (2.6) (e.g., the default Newton iteration), then a linear solver module header file will be required. The header files corresponding to the various Sundials-provided linear solver modules available for use with CVODE 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;
- cvode/cvode_diag.h, which is used with the CVDIAG diagonal linear solver module.

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 cvDiurnal_kry_p example (see [38]), 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 an ODE IVP. Most of the steps are independent of the NVECTOR, SUNMATRIX, SUNLINSOL, and SUNNONLINSOL implementations used. For the steps that are not, refer to Chapters 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 vector of initial values

To set the vector y0 of initial values, use the appropriate functions defined by the particular NVECTOR implementation.

For native SUNDIALS vector implementations (except the CUDA and RAJA-based ones), use a call of the form $y0 = N_VMake_***(..., ydata)$ 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.

4. Create CVODE object

Call cvode_mem = CVodeCreate(lmm) to create the CVODE memory block and to specify the linear multistep method. CVodeCreate returns a pointer to the CVODE memory structure. See §4.5.1 for details.

5. Initialize CVODE solver

Call CVodeInit(...) to provide required problem specifications, allocate internal memory for CVODE, and initialize CVODE. CVodeInit returns a flag, the value of which indicates either success or an illegal argument value. See §4.5.1 for details.

6. Specify integration tolerances

Call CVodeSStolerances(...) or CVodeSVtolerances(...) to specify either a scalar relative tolerance and scalar absolute tolerance, or a scalar relative tolerance and a vector of absolute tolerances, respectively. Alternatively, call CVodeWFtolerances 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 solve 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 $\S4.5.3$ 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 CVLS linear solver interface by attaching the linear solver object (and matrix object, if applicable) with the call (for details see §4.5.3):

```
ier = CVodeSetLinearSolver(...);
```

Alternately, if the CVODE-specific diagonal linear solver module, CVDIAG, is desired, initialize the linear solver module and attach it to CVODE with the call

```
ier = CVDiag(...);
```

11. Set optional inputs

Call CVodeSet* functions to change any optional inputs that control the behavior of CVODE from their default values. 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 = CVodeSetNonlinearSolver(cvode_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 CVodeInit if using the default nonlinear solver or after attaching a new nonlinear solver to CVODE, otherwise the optional inputs will be overridden by CVODE defaults. See Chapter 10 for more information on optional inputs.

15. Specify rootfinding problem (optional)

Call CVodeRootInit to initialize a rootfinding problem to be solved during the integration of the ODE system. See §4.5.5, and see §4.5.8.3 for relevant optional input calls.

16. Advance solution in time

For each point at which output is desired, call ier = CVode(cvode_mem, tout, yout, &tret, itask). Here itask specifies the return mode. The vector yout (which can be the same as the vector y0 above) will contain y(t). See §4.5.7 for details.

17. Get optional outputs

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

18. Deallocate memory for solution vector

Upon completion of the integration, deallocate memory for the vector y (or yout) by calling the appropriate destructor function defined by the NVECTOR implementation:

```
N_VDestroy(y);
```

19. Free solver memory

Call CVodeFree (&cvode_mem) to free the memory allocated by CVODE.

20. 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.

21. Free linear solver and matrix memory

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

22. 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.

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

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

4.5 User-callable functions

User Supp.

This section describes the CVODE functions that are called by the user to setup and then solve an IVP. Some of these are required. However, starting with §4.5.8, the functions listed involve optional inputs/outputs or restarting, and those paragraphs may be skipped for a casual use of CVODE. 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 CVODE 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 IVP solution is complete, as it frees the CVODE memory block created and allocated by the first two calls.

CVodeCreate

Description The function CVodeCreate instantiates a CVODE solver object and specifies the solution

method.

Arguments lmm (int) specifies the linear multistep method and must be one of two possible values:

CV_ADAMS or CV_BDF.

The recommended choices for 1mm are CV_ADAMS for nonstiff problems and CV_BDF for stiff problems. The default Newton iteration is recommended for stiff problems, and the fixed-point solver (previously referred to as the functional iteration in this guide) is recommended for nonstiff problems. For details on how to attach a different nonlinear solver module to CVODE see the description of CVodeSetNonlinearSolver.

Return value If successful, CVodeCreate returns a pointer to the newly created CVODE memory block (of type void *). Otherwise, it returns NULL.

F2003 Name FCVodeCreate

CVodeInit

Call flag = CVodeInit(cvode_mem, f, t0, y0);

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

internal memory, and initializes CVODE.

Arguments cvode_mem (void *) pointer to the CVODE memory block returned by CVodeCreate.

f (CVRhsFn) is the C function which computes the right-hand side function f in the ODE. This function has the form f(t, y, ydot, 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.

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

CV_SUCCESS The call to CVodeInit was successful.

CV_MEM_NULL The CVODE memory block was not initialized through a previous call to

CVodeCreate.

CV_MEM_FAIL A memory allocation request has failed.

CV_ILL_INPUT An input argument to CVodeInit has an illegal value.

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

tion.

F2003 Name FCVodeInit

CVodeFree

Call CVodeFree(&cvode_mem);

Description The function CVodeFree frees the memory allocated by a previous call to CVodeCreate.

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

Return value The function CVodeFree has no return value.

F2003 Name FCVodeFree

4.5.2 CVODE 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 CVodeInit.

CVodeSStolerances

Call flag = CVodeSStolerances(cvode_mem, reltol, abstol);

Description The function CVodeSStolerances specifies scalar relative and absolute tolerances.

Arguments cvode_mem (void *) pointer to the CVODE memory block returned by CVodeCreate.

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:

CV_SUCCESS The call to CVodeSStolerances was successful.

CV_MEM_NULL The CVODE memory block was not initialized through a previous call to CVodeCreate.

CV_NO_MALLOC The allocation function CVodeInit has not been called.

CV_ILL_INPUT One of the input tolerances was negative.

F2003 Name FCVodeSStolerances

CVodeSVtolerances

Call flag = CVodeSVtolerances(cvode_mem, reltol, abstol);

 $\label{prop:condense} \textbf{Description} \quad \text{The function $\tt CVodeSVtolerances specifies scalar relative tolerance and vector absolute}$

tolerances.

Arguments cvode_mem (void *) pointer to the CVODE memory block returned by CVodeCreate.

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:

CV_SUCCESS The call to CVodeSVtolerances was successful.

CV_MEM_NULL The CVODE memory block was not initialized through a previous call to

CVodeCreate.

CV_NO_MALLOC The allocation function CVodeInit has not been called.

 ${\tt CV_ILL_INPUT}$ 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 FCVodeSVtolerances

CVodeWFtolerances

Call flag = CVodeWFtolerances(cvode_mem, efun);

Description The function CVodeWFtolerances 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. (2.8).

Arguments cvode_mem (void *) pointer to the CVODE memory block returned by CVodeCreate.

efun (CVEwtFn) is the C function which defines the ewt vector (see §4.6.4).

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

CV_SUCCESS The call to CVodeWFtolerances was successful.

 $\begin{array}{ll} {\tt CV_MEM_NULL} & {\tt The\ CVODE\ memory\ block\ was\ not\ initialized\ through\ a\ previous\ call\ to} \\ {\tt CVodeCreate.} \end{array}$

CV_NO_MALLOC The allocation function CVodeInit has not been called.

F2003 Name FCVodeWFtolerances

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 1.0E-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 cvRoberts_dns in the CVODE package, and the discussion of it in the CVODE Examples document [38]. In that problem, the three components vary betwen 0 and 1, and have different noise levels; hence the abstol vector. It is impossible to give any general advice on abstol values, because the appropriate noise levels are completely problem-dependent. The user or modeler hopefully has some idea as to what those noise levels are.
- (3) Finally, it is important to pick all the tolerance values conservatively, because they control the error committed on each individual time step. The final (global) errors are some sort of accumulation of those per-step errors. A good rule of thumb is to reduce the tolerances by a factor of .01 from the actual desired limits on errors. So if you want .01% accuracy (globally), a good choice is $reltol = 10^{-6}$. 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 y returned by CVODE, with magnitude comparable to abstol or less, is equivalent to zero as far as the computation is concerned.
- (3) The user's right-hand side routine f should never change a negative value in the solution vector f to a non-negative value, as a "solution" to this problem. This can cause instability. If the f 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 f vector) for the purposes of computing f (f, f).
- (4) Positivity and non-negativity constraints on components can be enforced by use of the recoverable error return feature in the user-supplied right-hand side function. However, because this option involves some extra overhead cost, it 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 (2.6) (e.g., the default Newton iteration), there are two CVODE linear solver interfaces currently available for this task: CVLS and CVDIAG.

The first corresponds to the main linear solver interface in CVODE, that supports all valid SUN-LINSOL modules. Here, matrix-based SUNLINSOL modules utilize SUNMATRIX objects to store the approximate Jacobian matrix $J = \partial f/\partial y$, the Newton matrix $M = I - \gamma J$, and factorizations used throughout the solution process. Conversely, matrix-free sunlinsol modules instead use iterative methods to solve the Newton systems of equations, and only require the *action* of the matrix on a vector, Mv. With most of these methods, preconditioning can be done on the left only, the right only, on both the left and right, or not at all. The exceptions to this rule are specification of a preconditioning only and PCG that performs symmetric preconditioning. For the specification of a preconditioner, see the iterative linear solver sections in §4.5.8 and §4.6.

If preconditioning is done, user-supplied functions define linear operators corresponding to left and right preconditioner matrices P_1 and P_2 (either of which could be the identity matrix), such that the product P_1P_2 approximates the matrix $M = I - \gamma J$ of (2.7).

The CVDIAG linear solver interface supports a direct linear solver, that uses only a diagonal approximation to J.

To specify a generic linear solver to CVODE, after the call to CVodeCreate but before any calls to CVode, the user's program must create the appropriate SUNLinearSolver object and call the function CVodeSetLinearSolver, 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_SPECUMT, SUNLinSol_SPECUM, SUNLinSol_SPECUM, 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 CVODE via a call to CVodeSetLinearSolver. The first argument passed to this function is the CVODE memory pointer returned by CVodeCreate; the second argument is the desired SUNLINSOL object to use for solving linear 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 CVLS linear solver interface, linking it to the main CVODE integrator, and allows the user to specify additional parameters and routines pertinent to their choice of linear solver.

To instead specify the CVODE-specific diagonal linear solver interface, the user's program must call CVDiag, as documented below. The first argument passed to this function is the CVODE memory pointer returned by CVodeCreate.

CVodeSetLinearSolver

Call flag = CVodeSetLinearSolver(cvode_mem, LS, J);

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

Arguments cvode_mem (void *) pointer to the CVODE memory block.

LS (SUNLinearSolver) SUNLINSOL object to use for solving linear systems of the form (2.6).

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

CVLS_SUCCESS The CVLS initialization was successful.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_ILL_INPUT The CVLS interface is not compatible with the LS or J input objects or is incompatible with the current NVECTOR module.

CVLS_SUNLS_FAIL A call to the LS object failed.

CVLS_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).

When using sparse linear solvers, it is typically much more efficient to supply J so that it includes the full sparsity pattern of the Newton system matrices $M = I - \gamma J$, even if J itself has zeros in nonzero locations of I. The reasoning for this is that M is constructed in-place, on top of the user-specified values of J, so if the sparsity pattern in J is insufficient to store M then it will need to be resized internally by CVODE.

The previous routines CVDlsSetLinearSolver and CVSpilsSetLinearSolver 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.

 $F2003 \; \mathrm{Name} \; \; \mathsf{FCVodeSetLinearSolver}$

 ${\tt CVDiag}$

Call flag = CVDiag(cvode_mem);

Description The function CVDiag selects the CVDIAG linear solver.

The user's main program must include the cvode_diag.h header file.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

CVDIAG_SUCCESS The CVDIAG initialization was successful.

CVDIAG_MEM_NULL The cvode_mem pointer is NULL.

CVDIAG_ILL_INPUT The CVDIAG solver is not compatible with the current NVECTOR

module.

CVDIAG_MEM_FAIL A memory allocation request failed.

Notes

The CVDIAG solver is the simplest of all of the available CVODE linear solvers. The CVDIAG solver uses an approximate diagonal Jacobian formed by way of a difference quotient. The user does *not* have the option of supplying a function to compute an approximate diagonal Jacobian.

F2003 Name FCVDiag

4.5.4 Nonlinear solver interface function

By default CVODE uses the SUNNONLINSOL implementation of Newton's method defined by the SUNNONLINSOL_NEWTON module (see §10.3). To specify a different nonlinear solver in CVODE, the user's program must create a SUNNONLINSOL object by calling the appropriate constructor routine. The user must then attach the SUNNONLINSOL object by calling CVodeSetNonlinearSolver, as documented below.

When changing the nonlinear solver in CVODE, CVodeSetNonlinearSolver must be called after CVodeInit. If any calls to CVode have been made, then CVODE will need to be reinitialized by calling CVodeReInit to ensure that the nonlinear solver is initialized correctly before any subsequent calls to CVode.

The first argument passed to the routine CVodeSetNonlinearSolver is the CVODE memory pointer returned by CVodeCreate and the second argument is the SUNNONLINSOL object to use for solving the nonlinear system (2.4) or (2.5). A call to this function attaches the nonlinear solver to the main CVODE integrator.

CVodeSetNonlinearSolver

Call flag = CVodeSetNonlinearSolver(cvode_mem, NLS);

Description The function CVodeSetNonLinearSolver attaches a SUNNONLINSOL object (NLS) to

CVODE.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

NLS (SUNNonlinearSolver) SUNNONLINSOL object to use for solving nonlinear

systems (2.4) or (2.5).

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

CV_SUCCESS The nonlinear solver was successfully attached.

CV_MEM_NULL The cvode_mem pointer is NULL.

CV_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 FCVodeSetNonlinearSolver

4.5.5 Rootfinding initialization function

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

CVodeRootInit

Call flag = CVodeRootInit(cvode_mem, nrtfn, g);

Description The function CVodeRootInit specifies that the roots of a set of functions $g_i(t,y)$ are to

be found while the IVP is being solved.

Arguments cvode_mem (void *) pointer to the CVODE memory block returned by CVodeCreate.

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

g (CVRootFn) is the C function which defines the nrtfn functions $g_i(t,y)$

whose roots are sought. See §4.6.5 for details.

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

CV_SUCCESS The call to CVodeRootInit was successful.

CV_MEM_NULL The cvode_mem argument was NULL.

CV_MEM_FAIL A memory allocation failed.

CV_ILL_INPUT The function g is NULL, but nrtfn > 0.

Notes If a new IVP is to be solved with a call to CVodeReInit, where the new IVP has no

rootfinding problem but the prior one did, then call CVodeRootInit with nrtfn= 0.

 $F2003 \; Name \; FCVodeRootInit$

4.5.6 Projection initialization function

When solving an IVP with a constraint equation, CVODE has the capability to project the solution onto the constraint manifold after each time step. To activate the projection capability with a user-defined projection function, call the following set function:

CVodeSetProjFn

Call flag = CVodeSetProjFn(cvode_mem, proj);

Description The function CVodeSetProjFn enables or disables projection with a user-defined pro-

jection function.

Arguments cvode_mem (void *) is a pointer to the CVODE memory block returned by CVodeCreate.

proj (CVProjFn) is the C function which defines the projection. See §4.6.6 for

details.

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

CV_SUCCESS The call to CVodeSetProjFn was successful.

CV_MEM_NULL The cvode_mem argument was NULL.

CV_MEM_FAIL A memory allocation failed.

CV_ILL_INPUT The projection function is NULL or the method type is not CV_BDF.

Notes At this time projection is only supported with BDF methods.

If a new IVP is to be solved with a call to CVodeReInit, where the new IVP does not have a constraint equation but the prior one did, then call CVodeSetProjFrequency

with an input of 0 to disable projection.

F2003 Name FCVodeSetProjFn

4.5.7 CVODE solver function

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

CVode

Call flag = CVode(cvode_mem, tout, yout, &tret, itask);

Description The function CVode integrates the ODE over an interval in t.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

yout (N_Vector) the computed solution vector.

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

itask (int) a flag indicating the job of the solver for the next user step. The CV_NORMAL option causes the solver to take internal steps until it has reached or just passed the user-specified tout parameter. The solver then interpolates in order to return an approximate value of y(tout). The CV_ONE_STEP option tells the solver to take just one internal step and then return the

solution at the point reached by that step.

Return value CVode returns a vector yout and a corresponding independent variable value t = tret, such that yout is the computed value of y(t).

In CV_NORMAL mode (with no errors), tret will be equal to tout and yout = y(tout).

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

CV_SUCCESS CVode succeeded and no roots were found.

CV_TSTOP_RETURN CVode succeeded by reaching the stopping point specified through

the optional input function CVodeSetStopTime (see §4.5.8.1).

CV_ROOT_RETURN CVode succeeded and found one or more roots. In this case,

tret is the location of the root. If nrtfn > 1, call CVodeGetRootInfo

to see which g_i were found to have a root.

CV_MEM_NULL	The cvode_mem argument was NULL.			
CV_NO_MALLOC	The CVODE memory was not allocated by a call to CVodeInit.			
CV_ILL_INPUT	One of the inputs to CVode 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 CVodeCreate) failed to set the linear solver-specific lsolve field in cvode_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 error message for details.			
CV_T00_CLOSE	The initial time t_0 and the output time t_{out} are too close to each other and the user did not specify an initial step size.			
CV_TOO_MUCH_WORK	The solver took mxstep internal steps but still could not reach tout. The default value for mxstep is MXSTEP_DEFAULT = 500.			
CV_TOO_MUCH_ACC	The solver could not satisfy the accuracy demanded by the user for some internal step.			
CV_ERR_FAILURE	Either error test failures occurred too many times (MXNEF = 7) during one internal time step, or with $ h = h_{min}$.			
CV_CONV_FAILURE	Either convergence test failures occurred too many times (MXNCF = 10) during one internal time step, or with $ h = h_{min}$.			
CV_LINIT_FAIL	The linear solver interface's initialization function failed.			
CV_LSETUP_FAIL	The linear solver interface's setup function failed in an unrecoverable manner.			
CV_LSOLVE_FAIL	The linear solver interface's solve function failed in an unrecoverable manner.			
CV_CONSTR_FAIL	The inequality constraints were violated and the solver was unable to recover.			
CV_RHSFUNC_FAIL	The right-hand side function failed in an unrecoverable manner.			
CV_FIRST_RHSFUNC_FAIL	The right-hand side function had a recoverable error at the first call.			
CV_REPTD_RHSFUNC_ERR	Convergence test failures occurred too many times due to repeated recoverable errors in the right-hand side function. This flag will also be returned if the right-hand side function had repeated recoverable errors during the estimation of an initial step size.			
CV_UNREC_RHSFUNC_ERR	The right-hand function had a recoverable error, but no recovery was possible. This failure mode is rare, as it can occur only if the right-hand side function fails recoverably after an error test failed while at order one.			
CV_RTFUNC_FAIL	The rootfinding function failed.			
The vector yout can occupy the same space as the vector y0 of initial conditions that				
1 4 - OV - 1 - T 4				

 ${\rm Notes}$

The vector yout can occupy the same space as the vector yo of initial conditions that was passed to CVodeInit.

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

If a stop time is enabled (through a call to CVodeSetStopTime), then CVode 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 CVodeSetStopTime).

All failure return values are negative and so the test flag < 0 will trap all CVode failures.

On any error return in which one or more internal steps were taken by CVode, the returned values of tret and yout correspond to the farthest point reached in the integration. On all other error returns, tret and yout are left unchanged from the previous CVode return.

F2003 Name FCVode

4.5.8 Optional input functions

There are numerous optional input parameters that control the behavior of the CVODE solver. CVODE 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 CVODE which are then described in detail in the remainder of this section, begining with those for the main CVODE solver and continuing with those for the linear solver interfaces. Note that the diagonal linear solver module has no optional inputs. For the most casual use of CVODE, the reader can skip to §4.6.

We note that, on an error return, all of the optional input functions send an error message to the error handler function. All error return values are negative, so the test flag < 0 will catch all errors. Finally, a call to a CVodeSet*** 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 either of the functions CVodeSetErrFile or CVodeSetErrHandlerFn is to be called, that call should be first, in order to take effect for any later error message.

CVodeSetErrFile

Call flag = CVodeSetErrFile(cvode_mem, errfp);

Description The function CVodeSetErrFile specifies a pointer to the file where all CVODE messages

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

Arguments cvode_mem (void *) pointer to the CVODE memory block.

errfp (FILE *) pointer to output file.

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes The default value for errfp is stderr.

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

If CVodeSetErrFile 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 FCVodeSetErrFile

CVodeSetErrHandlerFn

Call flag = CVodeSetErrHandlerFn(cvode_mem, ehfun, eh_data);

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

used in handling error messages.

Arguments cvode_mem (void *) pointer to the CVODE memory block.



Table 4.2: Optional inputs for CVODE and CVLS

Optional input	Function name	Default					
CVODE main solver							
Pointer to an error file	CVodeSetErrFile	stderr					
Error handler function	CVodeSetErrHandlerFn	internal fn.					
User data	CVodeSetUserData	NULL					
Maximum order for BDF method	CVodeSetMaxOrd	5					
Maximum order for Adams method	CVodeSetMaxOrd	12					
Maximum no. of internal steps before t_{out}	CVodeSetMaxNumSteps	500					
Maximum no. of warnings for $t_n + h = t_n$	CVodeSetMaxHnilWarns	10					
Flag to activate stability limit detection	CVodeSetStabLimDet	SUNFALSE					
Initial step size	CVodeSetInitStep	estimated					
Minimum absolute step size	CVodeSetMinStep	0.0					
Maximum absolute step size	CVodeSetMaxStep	∞					
Value of t_{stop}	CVodeSetStopTime	undefined					
Maximum no. of error test failures	CVodeSetMaxErrTestFails	7					
Maximum no. of nonlinear iterations	CVodeSetMaxNonlinIters	3					
Maximum no. of convergence failures	CVodeSetMaxConvFails	10					
Coefficient in the nonlinear convergence test	CVodeSetNonlinConvCoef	0.1					
Inequality constraints on solution	CVodeSetConstraints	NULL					
Direction of zero-crossing	CVodeSetRootDirection	both					
Disable rootfinding warnings	CVodeSetNoInactiveRootWarn	none					
Flag to activate specialized fused kernels	CVodeSetUseIntegratorFusedKernels	SUNFALSE					
CVLS line	ear solver interface						
Linear solver setup frequency	CVodeSetLSetupFrequency	20					
Jacobian / preconditioner update frequency	CVodeSetJacEvalFrequency	51					
Jacobian function	CVodeSetJacFn	DQ					
Linear System function	CVodeSetLinSysFn	internal					
Enable or disable linear solution scaling	CVodeSetLinearSolutionScaling	on					
Jacobian-times-vector functions	CVodeSetJacTimes	NULL, DQ					
Jacobian-times-vector DQ RHS function	CVodeSetJacTimesRhsFn	NULL					
Preconditioner functions	CVodeSetPreconditioner	NULL, NULL					
Ratio between linear and nonlinear tolerances	CVodeSetEpsLin	0.05					
Newton linear solve tolerance conversion factor	CVodeSetLSNormFactor	vector length					

ehfun (CVErrHandlerFn) is the 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

CV_SUCCESS The function enfun and data pointer eh_data have been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes Error messages indicating that the CVODE solver memory is NULL will always be directed

to stderr.

F2003 Name FCVodeSetErrHandlerFn

CVodeSetUserData

Call flag = CVodeSetUserData(cvode_mem, user_data);

Description The function CVodeSetUserData specifies the user data block user_data and attaches

it to the main CVODE memory block.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

user_data (void *) pointer to the user data.

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_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

CVodeSetUserData must be made before the call to specify the linear solver.

F2003 Name FCVodeSetUserData

CVodeSetMonitorFn

Call flag = CVodeSetMonitorFn(cvode_mem, monitorfn);

Description The function CVodeSetMonitorFn specifies a user function, monitorfn, to be called at

some interval of successfully completed CVODE time steps.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

monitorfn (CVMonitorFn) user-supplied monitor function (NULL by default); a NULL

input will turn off monitoring

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes The frequency with which the monitor function is called can be set with the function

CVodeSetMonitorFrequency.

Modifying the solution in this function will result in undefined behavior. This function is only intended to be used for monitoring the integrator.

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 FCVodeSetMonitorFn



CVodeSetMonitorFrequency

Call flag = CVodeSetMonitorFrequency(cvode_mem, nst);

Description The function CVodeSetMonitorFrequency specifies the interval, measured in success-

fully completed CVODE time-steps, at which the monitor function should be called.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

nst (long int) number of successful steps inbetween calls to the monitor func-

tion (0 by default); a 0 input will turn off monitoring

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes The monitor function that will be called can be set with CVodeSetMonitorFn.

Modifying the solution in this function will result in undefined behavior. This function

is only intended to be used for monitoring the integrator.

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 FCVodeSetMonitorFrequency

CVodeSetMaxOrd

Call flag = CVodeSetMaxOrd(cvode_mem, maxord);

Description The function CVodeSetMaxOrd specifies the maximum order of the linear multistep

method.

Arguments cvode_mem (void *) pointer to the CVODE 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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

CV_ILL_INPUT The specified value maxord is ≤ 0 , or larger than its previous value.

Notes The default value is ADAMS_Q_MAX = 12 for the Adams-Moulton method and BDF_Q_MAX

= 5 for the BDF method. Since maxord affects the memory requirements for the internal

CVODE memory block, its value cannot be increased past its previous value.

An input value greater than the default will result in the default value.

 $F2003 \; \mathrm{Name} \; \; \mathsf{FCVodeSetMaxOrd}$

${\tt CVodeSetMaxNumSteps}$

Call flag = CVodeSetMaxNumSteps(cvode_mem, mxsteps);

Description The function CVodeSetMaxNumSteps specifies the maximum number of steps to be taken

by the solver in its attempt to reach the next output time.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

mxsteps (long int) maximum allowed number of steps.

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

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

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

F2003 Name FCVodeSetMaxNumSteps



CVodeSetMaxHnilWarns

Call flag = CVodeSetMaxHnilWarns(cvode_mem, mxhnil);

Description The function CVodeSetMaxHnilWarns specifies the maximum number of messages issued

by the solver warning that t + h = t on the next internal step.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

mxhnil (int) maximum number of warning messages (> 0).

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes The default value is 10. A negative value for mxhnil indicates that no warning messages

should be issued.

F2003 Name FCVodeSetMaxHnilWarns

CVodeSetStabLimDet

Call flag = CVodeSetstabLimDet(cvode_mem, stldet);

Description The function CVodeSetStabLimDet indicates if the BDF stability limit detection algo-

rithm should be used. See §2.4 for further details.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

stldet (booleantype) flag controlling stability limit detection (SUNTRUE = on;

SUNFALSE = off).

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

CV_ILL_INPUT The linear multistep method is not set to CV_BDF.

Notes The default value is SUNFALSE. If stldet = SUNTRUE when BDF is used and the method

order is greater than or equal to 3, then an internal function, CVsldet, is called to detect

a possible stability limit. If such a limit is detected, then the order is reduced.

F2003 Name FCVodeSetStabLimDet

CVodeSetInitStep

Call flag = CVodeSetInitStep(cvode_mem, hin);

Description The function CVodeSetInitStep specifies the initial step size.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

the default value.

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes By default, CVODE estimates the initial step size to be the solution h of the equation

 $||0.5h^2\ddot{y}||_{\text{WRMS}} = 1$, where \ddot{y} is an estimated second derivative of the solution at t0.

F2003 Name FCVodeSetInitStep

CVodeSetMinStep

Call flag = CVodeSetMinStep(cvode_mem, hmin);

Description The function CVodeSetMinStep specifies a lower bound on the magnitude of the step

size.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

hmin (realtype) minimum absolute value of the step size (≥ 0.0).

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

CV_ILL_INPUT Either hmin is nonpositive or it exceeds the maximum allowable step size.

Notes The default value is 0.0.

F2003 Name FCVodeSetMinStep

CVodeSetMaxStep

Call flag = CVodeSetMaxStep(cvode_mem, hmax);

Description The function CVodeSetMaxStep specifies an upper bound on the magnitude of the step

size.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

hmax (realtype) maximum absolute value of the step size (≥ 0.0).

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

 ${\tt CV_ILL_INPUT} \ \, {\tt Either} \ \, {\tt hmax} \ \, {\tt is} \ \, {\tt nonpositive} \ \, {\tt or} \ \, {\tt it} \ \, {\tt is} \ \, {\tt smaller} \ \, {\tt than} \ \, {\tt the} \ \, {\tt minimum} \ \, {\tt allowable}$

step size.

Notes Pass hmax = 0.0 to obtain the default value ∞ .

F2003 Name FCVodeSetMaxStep

CVodeSetStopTime

Call flag = CVodeSetStopTime(cvode_mem, tstop);

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

which the solution is not to proceed.

Arguments cvode_mem (void *) pointer to the CVODE 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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

CV_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 CVodeSetStopTime).

 $F2003\ \mathrm{Name}\ FCVodeSetStopTime$

CVodeSetMaxErrTestFails

Call flag = CVodeSetMaxErrTestFails(cvode_mem, maxnef);

Description The function CVodeSetMaxErrTestFails specifies the maximum number of error test

failures permitted in attempting one step.

Arguments cvode_mem (void *) pointer to the CVODE 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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes The default value is 7.

F2003 Name FCVodeSetMaxErrTestFails

CVodeSetMaxNonlinIters

Call flag = CVodeSetMaxNonlinIters(cvode_mem, maxcor);

Description The function CVodeSetMaxNonlinIters specifies the maximum number of nonlinear

solver iterations permitted per step.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

maxcor (int) maximum number of nonlinear solver iterations allowed per step (>0).

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

CV_MEM_FAIL The SUNNONLINSOL module is NULL.

Notes The default value is 3.

F2003 Name FCVodeSetMaxNonlinIters

CVodeSetMaxConvFails

Call flag = CVodeSetMaxConvFails(cvode_mem, maxncf);

 $\label{lem:convFails} \textbf{Description} \quad \text{The function $\tt CVodeSetMaxConvFails specifies the maximum number of nonlinear solver} \\$

convergence failures permitted during one step.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

per step (>0).

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes The default value is 10.

$F2003 \; \mathrm{Name} \;\; \mathsf{FCVodeSetMaxConvFails}$

CVodeSetNonlinConvCoef

Call flag = CVodeSetNonlinConvCoef(cvode_mem, nlscoef);

Description The function CVodeSetNonlinConvCoef specifies the safety factor used in the nonlinear

convergence test (see $\S 2.1$).

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes The default value is 0.1.

F2003 Name FCVodeSetNonlinConvCoef

CVodeSetConstraints

Call flag = CVodeSetConstraints(cvode_mem, constraints);

Description The function CVodeSetConstraints specifies a vector defining inequality constraints

for each component of the solution vector y.

Arguments cvode_mem (void *) pointer to the CVODE 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 $y_i \geq 0.0$.

-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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

CV_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. A NULL constraints vector will disable constraint checking.

F2003 Name FCVodeSetConstraints

CVodeSetUseIntegratorFusedKernels

Call flag = CVodeSetUseIntegratorFusedKernels(cvode_mem, onoff);

Description

The function CVodeSetUseIntegratorFusedKernels informs CVODE that it should use specialized fused kernels internally, if available. The specialized kernels may offer performance improvements for small problem sizes. Users should beware that these kernels can cause changes in the behavior of the integrator. By default, these kernels are not used. Must be called after CVodeInit.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

onoff (booleantype) boolean flag to turn on the specialized kernels (SUNTRUE), or to turn them off (SUNFALSE)

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes

SUNDIALS must be compiled appropriately for specialized kernels to be available. The CMake option SUNDIALS_BUILD_PACKAGE_FUSED_KERNELS must be set to ON when SUNDIALS is compiled. See the entry for this option in section A.1.2 for more requirements. Currently, the fused kernels are only supported when using CVODE with the NVECTOR_CUDA module 7.9.

4.5.8.2 Linear solver interface optional input functions

The mathematical explanation of the linear solver methods available to CVODE is provided in §2.1. We group the user-callable routines into four categories: general routines concerning the overall CVLs 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.

As discussed in §2.1, CVODE strives to reuse matrix and preconditioner data for as many solves as possible to amortize the high costs of matrix construction and factorization. To that end, CVODE provides user-callable routines to modify this behavior. Recall that the Newton system matrices are $M(t,y) = I - \gamma J(t,y)$, where the right-hand side function has Jacobian matrix $J(t,y) = \frac{\partial f(t,y)}{\partial x}$.

 $M(t,y) = I - \gamma J(t,y)$, where the right-hand side function has Jacobian matrix $J(t,y) = \frac{\partial f(t,y)}{\partial y}$. The matrix or preconditioner for M can only be updated within a call to the linear solver 'setup' routine. In general, the frequency with which this setup routine is called may be controlled with the msbp argument to CVodeSetLSetupFrequency. When this occurs, the validity of M for successive time steps intimately depends on whether the corresponding γ and J inputs remain valid.

At each call to the linear solver setup routine the decision to update M with a new value of γ , and to reuse or reevaluate Jacobian information, depends on several factors including:

- the success or failure of previous solve attempts,
- the success or failure of the previous time step attempts,
- the change in γ from the value used when constructing M, and
- the number of steps since Jacobian information was last evaluated.

The frequency with which to update Jacobian information can be controlled with the msbj argument to CVodeSetJacEvalFrequency. We note that this is only checked within calls to the linear solver setup routine, so values msbj < msbp do not make sense. For linear-solvers with user-supplied preconditioning the above factors are used to determine whether to recommend updating the Jacobian information in the preconditioner (i.e., whether to set jok to SUNFALSE in calling the user-supplied preconditioner setup function (see §4.6.11). For matrix-based linear solvers these factors determine whether the matrix $J(t,y) = \frac{\partial f(t,y)}{\partial y}$ should be updated (either with an internal finite difference approximation or a call to the user-supplied Jacobian function (see §4.6.7)); if not then the previous value is reused and the system matrix $M(t,y) \approx I - \gamma J(t,y)$ is recomputed using the current γ value.

CVodeSetLSetupFrequency

Arguments

Call retval = CVodeSetLSetupFrequency(cvode_mem, msbp);

Description The function CVodeSetLSetupFrequency specifies the frequency of calls to the linear solver setup function.

cvode_mem (void *) pointer to the CVODE memory block.

msbp (long int) the linear solver setup frequency.

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL. CV_ILL_INPUT The frequency msbp is negative.

Notes Positive values of msbp specify t

Positive values of msbp specify the linear solver setup frequency. For example, an input of 1 means the setup function will be called every time step while an input of 2 means it will be called every other time step. If msbp = 0, the default value of 20 will be used. Otherwise an error is returned.

F2003 Name FCVodeSetLSetupFrequency

CVodeSetJacEvalFrequency

Call retval = CVodeSetJacEvalFrequency(cvode_mem, msbj);

Description The function CVodeSetJacEvalFrequency specifies the frequency for recomputing the

Jacobian or recommending a preconditioner update.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

msbj (long int) the Jacobian re-computation or preconditioner update frequency.

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

CVLS_SUCCESS The optional value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS linear solver interface has not been initialized.

CVLS_ILL_INPUT The frequency msbj is negative.

Notes The Jacobian update frequency is only checked *within* calls to the linear solver setup routine, as such values of msbj < msbp will result in recomputing the Jacobian every

msbp steps. See CVodeSetLSetupFrequency for setting the linear solver setup frequency

msbp.

If msbj = 0, the default value of 51 will be used. Otherwise an error is returned.

This function must be called *after* the CVLS linear solver interface has been initialized through a call to CVodeSetLinearSolver.

F2003 Name FCVodeSetJacEvalFrequency

When using matrix-based linear solver modules, the CVLS solver interface needs a function to compute an approximation to the Jacobian matrix J(t,y) or the linear system $M=I-\gamma J$. The function to evaluate J(t,y) the must be of type CVLsJacFn. 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 CVLS solver. To specify a user-supplied Jacobian function jac, CVLS provides the function CVodeSetJacFn. The CVLS 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 CVodeSetUserData.

CVodeSetJacFn

Call flag = CVodeSetJacFn(cvode_mem, jac);

Description The function CVodeSetJacFn specifies the Jacobian approximation function to be used

for a matrix-based solver within the CVLS interface.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

jac (CVLsJacFn) user-defined Jacobian approximation function.

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

CVLS_SUCCESS The optional value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLs linear solver interface has not been initialized.

Notes This function must be called *after* the CVLS linear solver interface has been initialized through a call to CVodeSetLinearSolver.

By default, CVLS 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 CVLsJacFn is described in §4.6.7.

The previous routine CVDlsSetJacFn 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 FCVodeSetJacFn

To specify a user-supplied linear system function linsys, CVLS provides the function CVodeSetLinSysFn. The CVLS interface passes the pointer user_data to the linear system function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied linear system function, without using global data in the program. The pointer user_data may be specified through CVodeSetUserData.

CVodeSetLinSysFn

Call flag = CVodeSetLinSysFn(cvode_mem, linsys);

Description The function CVodeSetLinSysFn specifies the linear system approximation function to

be used for a matrix-based solver within the CVLS interface.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

linsys (CVLsLinSysFn) user-defined linear system approximation function.

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

CVLS_SUCCESS The optional value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS linear solver interface has not been initialized.

Notes This function must be called *after* the CVLS linear solver interface has been initialized through a call to CVodeSetLinearSolver.

By default, CVLS uses an internal linear system function leveraging the SUNMATRIX API to form the system $M = I - \gamma J$ using either an internal finite difference approximation or user-supplied function to compute the Jacobian. If linsys is NULL, this default function is used.

The function type CVLsLinSysFn is described in §4.6.8.

F2003 Name FCVodeSetLinSysFn

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 γ may not be current and, with BDF methods, a scaling factor is applied to the solution of the linear system to account for the lagged value of γ . See §9.4.1 for more details. The function CVodeSetLinearSolutionScaling can be used to disable this scaling when necessary, e.g., when providing a custom linear solver that updates the matrix using the current γ as part of the solve.

CVodeSetLinearSolutionScaling

Call flag = CVodeSetLinearSolutionScaling(cvode_mem, onoff);

Description The function CVodeSetLinearSolutionScaling enables or disables scaling the linear system solution to account for a change in γ in the linear system. For more details see

ξ9.4.1.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

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

CVLS_SUCCESS The flag value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS linear solver interface has not been initialized.

CVLS_ILL_INPUT The attached linear solver is not matrix-based or the linear multistep method type is not BDF.

Notes

This function must be called *after* the CVLS linear solver interface has been initialized through a call to CVodeSetLinearSolver.

By default scaling is enabled with matrix-based linear solvers when using BDF methods.

F2003 Name FCVodeSetLinearSolutionScaling

When using matrix-free linear solver modules, the CVLS 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 CVLS interface.

A user-defined Jacobian-vector product function must be of type CVLsJacTimesVecFn and can be specified through a call to CVodeSetJacTimes (see §4.6.9 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 §4.6.10 for specification details). The pointer user_data received through CVodeSetUserData (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.

CVodeSetJacTimes

Call flag = CVodeSetJacTimes(cvode_mem, jtsetup, jtimes);

Description The function CVodeSetJacTimes specifies the Jacobian-vector setup and product function.

tions.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

jtsetup (CVLsJacTimesSetupFn) user-defined Jacobian-vector setup function. Pass NULL if no setup is necessary.

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

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

CVLS_SUCCESS The optional value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS linear solver has not been initialized.

CVLS_SUNLS_FAIL An error occurred when setting up the system matrix-times-vector routines in the SUNLINSOL object used by the CVLS 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 CVLS linear solver interface has been initialized through a call to CVodeSetLinearSolver.

The function type CVLsJacTimesSetupFn is described in $\S4.6.10$.

The function type CVLsJacTimesVecFn is described in §4.6.9.

The previous routine CVSpilsSetJacTimes 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 FCVodeSetJacTimes

When using the internal difference quotient the user may optionally supply an alternative right-hand side function for use in the Jacobian-vector product approximation by calling CVodeSetJacTimesRhsFn. The alternative right-hand side function should compute a suitable (and differentiable) approximation to the right-hand side function provided to CVodeInit. For example, as done in [26], the alternative function may use lagged values when evaluating a nonlinearity in the right-hand side to avoid differencing a potentially non-differentiable factor.

CVodeSetJacTimesRhsFn

Call flag = CVodeSetJacTimesRhsFn(cvode_mem, jtimesRhsFn);

Description The function CVodeSetJacTimesRhsFn specifies an alternative ODE right-hand side

function for use in the internal Jacobian-vector product difference quotient approxima-

tion.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

jtimesRhsFn (CVRhsFn) is the C function which computes the alternative ODE right-hand side function to use in Jacobian-vector product difference quotient

approximations. This function has the form f(t, y, ydot, user_data)

(for full details see $\S4.6.1$).

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

CVLS_SUCCESS The optional value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS linear solver has not been initialized.

CVLS_ILL_INPUT The internal difference quotient approximation is disabled.

Notes The default is to use the right-hand side function provided to CVodeInit in the internal difference quotient. If the input right-hand side function is NULL, the default is used.

This function must be called after the CVLS linear solver interface has been initialized

through a call to CVodeSetLinearSolver.

F2003 Name FCVodeSetJacTimesRhsFn

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 CVODE using the function CVodeSetPreconditioner. 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. The user data pointer received through CVodeSetUserData (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 CVLS interface requires that iterative linear solvers stop when the norm of the preconditioned residual satisfies

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

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

CVodeSetPreconditioner

Call flag = CVodeSetPreconditioner(cvode_mem, psetup, psolve);

Description The function CVodeSetPreconditioner specifies the preconditioner setup and solve

functions.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

psetup (CVLsPrecSetupFn) user-defined preconditioner setup function. Pass NULL

if no setup is necessary.

psolve (CVLsPrecSolveFn) user-defined preconditioner solve function.

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

CVLS_SUCCESS The optional values have been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS linear solver has not been initialized.

CVLS_SUNLS_FAIL An error occurred when setting up preconditioning in the SUNLINSOL object used by the CVLS interface.

Notes

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

This function must be called *after* the CVLS linear solver interface has been initialized through a call to CVodeSetLinearSolver.

The function type CVLsPrecSolveFn is described in §4.6.11.

The function type CVLsPrecSetupFn is described in §4.6.12.

The previous routine CVSpilsSetPreconditioner 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 FCVodeSetPreconditioner

CVodeSetEpsLin

Call flag = CVodeSetEpsLin(cvode_mem, eplifac);

convergence test constant is reduced from the nonlinear solver test constant.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

eplifac (realtype) linear convergence safety factor (≥ 0.0).

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

CVLS_SUCCESS The optional value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS linear solver has not been initialized.

CVLS_ILL_INPUT The factor eplifac is negative.

Notes The default value is 0.05.

This function must be called *after* the CVLS linear solver interface has been initialized through a call to CVodeSetLinearSolver.

If eplifac = 0.0 is passed, the default value is used.

The previous routine CVSpilsSetEpsLin 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 FCVodeSetEpsLin

CVodeSetLSNormFactor

Call flag = CVodeSetLSNormFactor(cvode_mem, nrmfac);

 $\label{prop:converting} \textbf{Description} \quad \textbf{The function $\tt CVodeSetLSNormFactor specifies the factor to use when converting from the properties of the factor of the converting from the converting from the converting of the converting from the c$

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 cvode_mem (void *) pointer to the CVODE 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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes This function must be called after the CVLS linear solver interface has been initialized

through a call to CVodeSetLinearSolver.

Prior to the introduction of N_VGetLength in SUNDIALS v5.0.0 (CVODE v5.0.0) the value of nrmfac was computed using the vector dot product i.e., the nrmfac < 0 case.

F2003 Name FCVodeSetLSNormFactor

4.5.8.3 Rootfinding optional input functions

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

CVodeSetRootDirection

Call flag = CVodeSetRootDirection(cvode_mem, rootdir);

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

located and returned.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

rootdir (int *) state array of length nrtfn, the number of root functions g_i , as spec-

ified in the call to the function CVodeRootInit. A value of 0 for rootdir[i] indicates that crossing in either direction for g_i should be reported. 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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

CV_ILL_INPUT rootfinding has not been activated through a call to CVodeRootInit.

Notes The default behavior is to monitor for both zero-crossing directions.

F2003 Name FCVodeSetRootDirection

CVodeSetNoInactiveRootWarn

Call flag = CVodeSetNoInactiveRootWarn(cvode_mem);

Description The function CVodeSetNoInactiveRootWarn disables issuing a warning if some root

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

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes CVODE 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), CVODE will issue a warning which can be disabled with this optional

input function.

F2003 Name FCVodeSetNoInactiveRootWarn

4.5.8.4 Projection optional input functions

The following functions can be called to set optional inputs to control the projection when solving an IVP with constraints.

CVodeSetProjErrEst

Call flag = CVodeSetProjErrEst(cvode_mem, onoff);

Description The function CVodeSetProjErrEst enables or disables projection of the error estimate

by the projection function.

Arguments cvode_mem (void *) is a pointer to the CVODE memory block.

onoff (booleantype) is a flag indicating if error projection should be enabled

(SUNTRUE, default) or disabled (SUNFALSE).

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

CV_PROJ_MEM_NULL The projection memory is NULL i.e., the projection functionality has

not been enabled.

F2003 Name FCVodeSetProjErrEst

CVodeSetProjFrequency

Call flag = CVodeSetProjFrequency(cvode_mem, freq);

Description The function CVodeSetProjFrequency specifies the frequency with which the projection

is performed.

Arguments cvode_mem (void *) is a pointer to the CVODE memory block.

freq (long int) is the frequency with which to perform the projection. The

default is 1 (project every step), a value of 0 will disable projection, and a

value < 0 will restore the default.

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

CV_PROJ_MEM_NULL The projection memory is NULL i.e., the projection functionality has

not been enabled.

F2003 Name FCVodeSetProjFrequency

CVodeSetMaxNumProjFails

Call flag = CVodeSetMaxNumProjFails(cvode_mem, max_fails);

Description The function CVodeSetMaxNumProjFails specifies the maximum number of projection

failures in a step attempt before an unrecoverable error is returned.

Arguments cvode_mem (void *) is a pointer to the CVODE memory block.

max_fails (int) is the maximum number of projection failures. The default is 10 and

an input value < 1 will restore the default.

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

 ${\tt CV_PROJ_MEM_NULL} \ \ {\tt The \ projection \ memory \ is \ NULL \ i.e., \ the \ projection \ functionality \ has}$

not been enabled.

F2003 Name FCVodeSetMaxNumProjFails

CVodeSetEpsProj

Call flag = CVodeSetEpsProj(cvode_mem, eps);

Description The function CVodeSetEpsProj specifies the tolerance for the nonlinear constrained

least squares problem solved by the projection function.

Arguments cvode_mem (void *) is a pointer to the CVODE memory block.

eps (realtype) is the tolerance (default 0.1) for the the nonlinear constrained

least squares problem solved by the projection function. A value ≤ 0 will

restore the default.

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

CV_PROJ_MEM_NULL The projection memory is NULL i.e., the projection functionality has

not been enabled.

F2003 Name FCVodeSetEpsProj

CVodeSetProjFailEta

Call flag = CVodeSetProjFailEta(cvode_mem, eta);

Description The function CVodeSetProjFailEta specifies the time step reduction factor to apply

on a projection function failure.

Arguments cvode_mem (void *) is a pointer to the CVODE memory block.

eps (realtype) is the time step reduction factor to apply on a projection func-

tion failure (default 0.25). A value ≤ 0 or > 1 will restore the default.

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

 ${\tt CV_PROJ_MEM_NULL} \ \ {\tt The \ projection \ memory \ is \ NULL \ i.e., \ the \ projection \ functionality \ has}$

not been enabled.

 $F2003 \ \mathrm{Name}$ FCVodeSetProjFailEta

4.5.9 Interpolated output function

An optional function CVodeGetDky is available to obtain additional output values. This function should only be called after a successful return from CVode as it provides interpolated values either of y or of its derivatives (up to the current order of the integration method) interpolated to any value of t in the last internal step taken by CVODE.

The call to the CVodeGetDky function has the following form:

CVodeGetDky

Call flag = CVodeGetDky(cvode_mem, t, k, dky);

Description The function CVodeGetDky computes the k-th derivative of the function y at time t, i.e.

 $d^{(k)}y/dt^{(k)}(t)$, where $t_n - h_u \le t \le t_n$, t_n denotes the current internal time reached, and h_u is the last internal step size successfully used by the solver. The user may request k

 $=0,1,\ldots,q_u,$ where q_u is the current order (optional output qlast).

Arguments cvode_mem (void *) pointer to the CVODE memory block.

t (realtype) the value of the independent variable at which the derivative is to be evaluated.

k (int) the derivative order requested.

dky (N_Vector) vector containing the derivative. This vector must be allocated

by the user.

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

CV_SUCCESS CVodeGetDky succeeded.

CV_BAD_K k is not in the range $0,1,\ldots,q_u$. CV_BAD_T t is not in the interval $[t_n-h_u,t_n]$.

CV_BAD_DKY The dky argument was NULL.

CV_MEM_NULL The cvode_mem argument was NULL.

Notes It is only legal to call the function CVodeGetDky after a successful return from CVode.

See CVodeGetCurrentTime, CVodeGetLastOrder, and CVodeGetLastStep in the next

section for access to t_n , q_u , and h_u , respectively.

F2003 Name FCVodeGetDky

4.5.10 Optional output functions

CVODE provides an extensive set of functions that can be used to obtain solver performance information. Table 4.3 lists all optional output functions in CVODE, 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 CVODE solver is in doing its job. For example, the counters nsteps and nfevals 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 CVODE, CVLS, and CVDIAG

Optional output	Function name		
CVODE main solv	ver		
Size of CVODE real and integer workspaces	CVodeGetWorkSpace		
Cumulative number of internal steps	CVodeGetNumSteps		
No. of calls to r.h.s. function	CVodeGetNumRhsEvals		
No. of calls to linear solver setup function	CVodeGetNumLinSolvSetups		
No. of local error test failures that have occurred	CVodeGetNumErrTestFails		
Order used during the last step	CVodeGetLastOrder		
Order to be attempted on the next step	CVodeGetCurrentOrder		
No. of order reductions due to stability limit detection	CVodeGetNumStabLimOrderReds		
Actual initial step size used	CVodeGetActualInitStep		
Step size used for the last step	CVodeGetLastStep		
Step size to be attempted on the next step	CVodeGetCurrentStep		
Current internal time reached by the solver	CVodeGetCurrentTime		
Suggested factor for tolerance scaling	CVodeGetTolScaleFactor		
Error weight vector for state variables	CVodeGetErrWeights		
Estimated local error vector	CVodeGetEstLocalErrors		
No. of nonlinear solver iterations	CVodeGetNumNonlinSolvIters		
No. of nonlinear convergence failures	CVodeGetNumNonlinSolvConvFails		
All CVODE integrator statistics	CVodeGetIntegratorStats		
CVODE nonlinear solver statistics	CVodeGetNonlinSolvStats		
Array showing roots found	CVodeGetRootInfo		
No. of calls to user root function	CVodeGetNumGEvals		
Name of constant associated with a return flag	CVodeGetReturnFlagName		
CVLS linear solver interface			
Size of real and integer workspaces	CVodeGetLinWorkSpace		
No. of Jacobian evaluations	CVodeGetNumJacEvals		
No. of r.h.s. calls for finite diff. Jacobian[-vector] evals.	CVodeGetNumLinRhsEvals		
No. of linear iterations	CVodeGetNumLinIters		
No. of linear convergence failures	CVodeGetNumLinConvFails		
No. of preconditioner evaluations	CVodeGetNumPrecEvals		
No. of preconditioner solves	CVodeGetNumPrecSolves		
No. of Jacobian-vector setup evaluations	CVodeGetNumJTSetupEvals		
No. of Jacobian-vector product evaluations	CVodeGetNumJtimesEvals		
Get all linear solver statistics in one function call	CVodeGetLinSolvStats		
Last return from a linear solver function	${\tt CVodeGetLastLinFlag}$		
Name of constant associated with a return flag	CVodeGetLinReturnFlagName		
CVDIAG linear solver interface			
Size of CVDIAG real and integer workspaces	CVDiagGetWorkSpace		
No. of r.h.s. calls for finite diff. Jacobian evals.	CVDiagGetNumRhsEvals		
Last return from a CVDIAG function	CVDiagGetLastFlag		
Name of constant associated with a return flag	CVDiagGetReturnFlagName		

SUNDIALSGetVersionNumber

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

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

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

Arguments major (int) SUNDIALS release major version number.

minor (int) SUNDIALS release minor version number.

patch (int) SUNDIALS release patch version number.

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

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

Return value If successful, SUNDIALSGetVersionNumber returns 0 and the major, minor, patch, and

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

character array is too short).

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

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

characters in the label array are removed.

4.5.10.2 Main solver optional output functions

CVODE 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 CVODE memory block (a suggested tolerance scaling factor, the error weight vector, and the vector of estimated local errors). Functions are also provided to extract statistics related to the performance of the CVODE nonlinear solver used. As a convenience, additional information extraction functions provide the optional outputs in groups. These optional output functions are described next.

CVodeGetWorkSpace

Call flag = CVodeGetWorkSpace(cvode_mem, &lenrw, &leniw);

Description The function CVodeGetWorkSpace returns the CVODE real and integer workspace sizes.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

lenrw (long int) the number of realtype values in the CVODE workspace.

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

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

CV_SUCCESS The optional output values have been successfully set.

CV_MEM_NULL The cvode_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 §4.5.5), the actual size of the real workspace, in realtype words, is given by the following:

- base value: lenrw = $96 + (maxord+5) * N_r + 3*nrtfn;$
- using CVodeSVtolerances: lenrw = lenrw $+N_r$;
- with constraint checking (see CVodeSetConstraints): lenrw = lenrw $+N_r$;

where 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 = $40 + (maxord+5) * N_i + nrtfn;$
- using CVodeSVtolerances: leniw = leniw $+N_i$;

• with constraint checking: $lenrw = lenrw + N_i$;

where N_i is the number of integer words in one N_Vector (= 1 for NVECTOR_SERIAL and 2*npes for NVECTOR_PARALLEL and npes processors).

For the default value of maxord, no rootfinding, no constraints, and without using CVodeSVtolerances, these lengths are given roughly by:

- For the Adams method: lenrw = 96 + 17N and leniw = 57
- For the BDF method: lenrw = 96 + 10N and leniw = 50

F2003 Name FCVodeGetWorkSpace

CVodeGetNumSteps

Call flag = CVodeGetNumSteps(cvode_mem, &nsteps);

Description The function CVodeGetNumSteps returns the cumulative number of internal steps taken

by the solver (total so far).

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

 $F2003 \; \mathrm{Name} \; \; \mathsf{FCVodeGetNumSteps}$

CVodeGetNumRhsEvals

Call flag = CVodeGetNumRhsEvals(cvode_mem, &nfevals);

Description The function CVodeGetNumRhsEvals returns the number of calls to the user's right-hand

side function.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

nfevals (long int) number of calls to the user's f function.

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes The nfevals value returned by CVodeGetNumRhsEvals does not account for calls made

to f by a linear solver or preconditioner module.

F2003 Name FCVodeGetNumRhsEvals

CVodeGetNumLinSolvSetups

Call flag = CVodeGetNumLinSolvSetups(cvode_mem, &nlinsetups);

Description The function CVodeGetNumLinSolvSetups returns the number of calls made to the

linear solver's setup function.

Arguments cvode_mem (void *) pointer to the CVODE 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

 ${\tt CV_SUCCESS}$. The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

F2003 Name FCVodeGetNumLinSolvSetups

CVodeGetNumErrTestFails

Call flag = CVodeGetNumErrTestFails(cvode_mem, &netfails);

Description The function CVodeGetNumErrTestFails returns the number of local error test failures

that have occurred.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

netfails (long int) number of error test failures.

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

F2003 Name FCVodeGetNumErrTestFails

CVodeGetLastOrder

Call flag = CVodeGetLastOrder(cvode_mem, &qlast);

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

last internal step.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

F2003 Name FCVodeGetLastOrder

CVodeGetCurrentOrder

Call flag = CVodeGetCurrentOrder(cvode_mem, &qcur);

Description The function CVodeGetCurrentOrder returns the integration method order to be used

on the next internal step.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

F2003 Name FCVodeGetCurrentOrder

CVodeGetLastStep

Call flag = CVodeGetLastStep(cvode_mem, &hlast);

Description The function CVodeGetLastStep returns the integration step size taken on the last

internal step.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

F2003 Name FCVodeGetLastStep

CVodeGetCurrentStep

Call flag = CVodeGetCurrentStep(cvode_mem, &hcur);

Description The function CVodeGetCurrentStep returns the integration step size to be attempted

on the next internal step.

Arguments cvode_mem (void *) pointer to the CVODE 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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

F2003 Name FCVodeGetCurrentStep

CVodeGetActualInitStep

Call flag = CVodeGetActualInitStep(cvode_mem, &hinused);

Description The function CVodeGetActualInitStep returns the value of the integration step size

used on the first step.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

hinused (realtype) actual value of initial step size.

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes Even if the value of the initial integration step size was specified by the user through

a call to CVodeSetInitStep, this value might have been changed by CVODE to ensure that the step size is within the prescribed bounds $(h_{\min} \le h_0 \le h_{\max})$, or to satisfy the

local error test condition.

F2003 Name FCVodeGetActualInitStep

CVodeGetCurrentTime

Call flag = CVodeGetCurrentTime(cvode_mem, &tcur);

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

solver.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

tcur (realtype) current internal time reached.

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

 $F2003\ Name\ FCVodeGetCurrentTime$

CVodeGetNumStabLimOrderReds

Call flag = CVodeGetNumStabLimOrderReds(cvode_mem, &nslred);

Description The function CVodeGetNumStabLimOrderReds returns the number of order reductions

dictated by the BDF stability limit detection algorithm (see $\S 2.4$).

Arguments cvode_mem (void *) pointer to the CVODE memory block.

nslred (long int) number of order reductions due to stability limit detection.

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes If the stability limit detection algorithm was not initialized (CVodeSetStabLimDet was

not called), then nslred = 0.

F2003 Name FCVodeGetNumStabLimOrderReds

CVodeGetTolScaleFactor

Call flag = CVodeGetTolScaleFactor(cvode_mem, &tolsfac);

Description The function CVodeGetTolScaleFactor 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 cvode_mem (void *) pointer to the CVODE memory block.

tolsfac (realtype) suggested scaling factor for user-supplied tolerances.

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

F2003 Name FCVodeGetTolScaleFactor

CVodeGetErrWeights

Call flag = CVodeGetErrWeights(cvode_mem, eweight);

Description The function CVodeGetErrWeights returns the solution error weights at the current

time. These are the reciprocals of the W_i given by (2.8).

Arguments cvode_mem (void *) pointer to the CVODE memory block.

eweight (N_Vector) solution error weights at the current time.

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes The user must allocate memory for eweight.

F2003 Name FCVodeGetErrWeights

CVodeGetEstLocalErrors

Call flag = CVodeGetEstLocalErrors(cvode_mem, ele);

Description The function CVodeGetEstLocalErrors returns the vector of estimated local errors.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

ele (N_Vector) estimated local errors.

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes The user must allocate memory for ele.

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

The ele vector, together with the eweight vector from CVodeGetErrWeights, 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 Name FCVodeGetEstLocalErrors

CVodeGetIntegratorStats

Call flag = CVodeGetIntegratorStats(cvode_mem, &nsteps, &nfevals, &nlinsetups, &netfails, &qlast, &qcur, &hinused, &hlast, &hcur, &tcur);

Description The function CVodeGetIntegratorStats returns the CVODE integrator statistics as a group.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

nsteps (long int) number of steps taken by CVODE.
nfevals (long int) number of calls to the user's f function.

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

netfails (long int) number of error test failures.

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

qcur (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

CV_SUCCESS the optional output values have been successfully set.

CV_MEM_NULL the cvode_mem pointer is NULL.

F2003 Name FCVodeGetIntegratorStats

CVodeGetNumNonlinSolvIters

Call flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nniters);

Description The function CVodeGetNumNonlinSolvIters returns the number of nonlinear iterations

performed.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

nniters (long int) number of nonlinear iterations performed.

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

CV_SUCCESS The optional output values have been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

CV_MEM_FAIL The SUNNONLINSOL module is NULL.

F2003 Name FCVodeGetNumNonlinSolvIters

CVodeGetNumNonlinSolvConvFails

Call flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &nncfails);

Description The function CVodeGetNumNonlinSolvConvFails returns the number of nonlinear con-

vergence failures that have occurred.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

nncfails (long int) number of nonlinear convergence failures.

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

F2003 Name FCVodeGetNumNonlinSolvConvFails

CVodeGetNonlinSolvStats

Call flag = CVodeGetNonlinSolvStats(cvode_mem, &nniters, &nncfails);

Description The function CVodeGetNonlinSolvStats returns the CVODE nonlinear solver statistics

as a group.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

CV_MEM_FAIL The SUNNONLINSOL module is NULL.

F2003 Name FCVodeGetNonlinSolvStats

CVodeGetReturnFlagName

Description The function CVodeGetReturnFlagName returns the name of the CVODE constant cor-

responding to flag.

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

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

F2003 Name FCVodeGetReturnFlagName

4.5.10.3 Rootfinding optional output functions

There are two optional output functions associated with rootfinding.

CVodeGetRootInfo

Call flag = CVodeGetRootInfo(cvode_mem, rootsfound);

Description The function CVodeGetRootInfo returns an array showing which functions were found

to have a root.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

rootsfound (int *) array of length nrtfn with the indices of the user functions g_i found to have a root. For $i = 0, \ldots, \text{nrtfn}-1$, 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:

CV_SUCCESS The optional output values have been successfully set.

CV_MEM_NULL The cvode_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 q_i .

The user must allocate memory for the vector rootsfound.

F2003 Name FCVodeGetRootInfo



CVodeGetNumGEvals

Call flag = CVodeGetNumGEvals(cvode_mem, &ngevals);

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

user-supplied root function g.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

F2003 Name FCVodeGetNumGEvals

4.5.10.4 Projection optional output functions

The following optional output functions are available for retrieving information and statistics related the projection when solving and IVP with constraints.

${\tt CVodeGetNumProjEvals}$

Call flag = CVodeGetNumProjEvals(cvode_mem, nproj);

Description The function CVodeGetNumProjEvals returns the current total number of projection

evaluations.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

nproj (long int *) the number of calls to the projection function.

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

CV_PROJ_MEM_NULL The projection memory is NULL i.e., the projection functionality has

not been enabled.

F2003 Name FCVodeGetNumProjEvals

CVodeGetNumProjFails

Call flag = CVodeGetNumProjFails(cvode_mem, npfails);

Description The function CVodeGetNumProjFails returns the current total number of projection

evaluation failures.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

npfails (long int *) the number of projection failures.

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

 ${\tt CV_PROJ_MEM_NULL \ The \ projection \ memory \ is \ NULL \ i.e., \ the \ projection \ functionality \ has}$

not been enabled.

F2003 Name FCVodeGetNumProjFails

4.5.10.5 CVLS linear solver interface optional output functions

The following optional outputs are available from the CVLS modules: workspace requirements, number of calls to the Jacobian routine, number of calls to the right-hand side 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 a linear solver 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).

${\tt CVodeGetLinWorkSpace}$

Call flag = CVodeGetLinWorkSpace(cvode_mem, &lenrwLS, &leniwLS);

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

used by the CVLS linear solver interface.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

lenrwLS (long int) the number of realtype values in the CVLS workspace.

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

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

CVLS_SUCCESS The optional output values have been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS 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 CVLS is not included in this report.

The previous routines CVDlsGetWorkspace and CVSpilsGetWorkspace 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 FCVodeGetLinWorkSpace

CVodeGetNumJacEvals

Call flag = CVodeGetNumJacEvals(cvode_mem, &njevals);

Description The function CVodeGetNumJacEvals returns the number of calls made to the CVLS

Jacobian approximation function.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

njevals (long int) the number of calls to the Jacobian function.

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

CVLS_SUCCESS The optional output value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS linear solver has not been initialized.

Notes The previous routine CVDlsGetNumJacEvals 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 FCVodeGetNumJacEvals

CVodeGetNumLinRhsEvals

Call flag = CVodeGetNumLinRhsEvals(cvode_mem, &nfevalsLS);

Description The function CVodeGetNumLinRhsEvals returns the number of calls made to the user-

supplied right-hand side function due to the finite difference Jacobian approximation or

finite difference Jacobian-vector product approximation.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

nfevalsLS (long int) the number of calls made to the user-supplied right-hand side

function.

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

CVLS_SUCCESS The optional output value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS linear solver has not been initialized.

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

functions is used.

The previous routines CVDlsGetNumRhsEvals and CVSpilsGetNumRhsEvals 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 FCVodeGetNumLinRhsEvals

CVodeGetNumLinIters

Call flag = CVodeGetNumLinIters(cvode_mem, &nliters);

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

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

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

CVLS_SUCCESS The optional output value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS linear solver has not been initialized.

Notes The previous routine CVSpilsGetNumLinIters 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 FCVodeGetNumLinIters

CVodeGetNumLinConvFails

Call flag = CVodeGetNumLinConvFails(cvode_mem, &nlcfails);

Description The function CVodeGetNumLinConvFails returns the cumulative number of linear con-

vergence failures.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

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

CVLS_SUCCESS The optional output value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS linear solver has not been initialized.

Notes The previous routine CVSpilsGetNumConvFails 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 FCVodeGetNumLinConvFails

CVodeGetNumPrecEvals

Call flag = CVodeGetNumPrecEvals(cvode_mem, &npevals);

Description The function CVodeGetNumPrecEvals returns the number of preconditioner evaluations,

i.e., the number of calls made to psetup with jok = SUNFALSE.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

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

CVLS_SUCCESS The optional output value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS linear solver has not been initialized.

Notes The previous routine CVSpilsGetNumPrecEvals 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 FCVodeGetNumPrecEvals

CVodeGetNumPrecSolves

Call flag = CVodeGetNumPrecSolves(cvode_mem, &npsolves);

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

the preconditioner solve function, psolve.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

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

CVLS_SUCCESS The optional output value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS linear solver has not been initialized.

Notes The previous routine CVSpilsGetNumPrecSolves 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 FCVodeGetNumPrecSolves

CVodeGetNumJTSetupEvals

Call flag = CVodeGetNumJTSetupEvals(cvode_mem, &njtsetup);

Description The function CVodeGetNumJTSetupEvals returns the cumulative number of calls made

to the Jacobian-vector setup function jtsetup.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

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

CVLS_SUCCESS The optional output value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS linear solver has not been initialized.

Notes The previous routine CVSpilsGetNumJTSetupEvals 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.

 $F2003 \; \mathrm{Name} \; \; \mathsf{FCVodeGetNumJTSetupEvals}$

CVodeGetNumJtimesEvals

Call flag = CVodeGetNumJtimesEvals(cvode_mem, &njvevals);

Description The function CVodeGetNumJtimesEvals returns the cumulative number of calls made

to the Jacobian-vector function jtimes.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

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

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

CVLS_SUCCESS The optional output value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS linear solver has not been initialized.

Notes The previous routine CVSpilsGetNumJtimesEvals 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 FCVodeGetNumJtimesEvals

CVodeGetLinSolvStats

Call flag = CVodeGetLinSolvStats(cvode_mem, &njevals, &nfevalsLS, &nliters, &nlcfails, &npeval

Description The function CVodeGetLinSolvStats returns CVODE linear solver statistics.

Arguments cvode_mem - (void *) the CVodeMem structure

njevals - (long int*) the current number of calls to the Jacobian function

nfevalsLS - (long int*) the current number of calls made to the user-supplied right-

hand side function by the linear solver

nliters - (long int*) the current number of linear iterations

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

npevals - (long int*) the current number of calls to psetup
npsolves - (long int*) the current number of calls to psolve
njtsetup - (long int*) the current number of calls to jtsetup
njtimes - (long int*) the current number of calls to jtimes

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

CVLS_SUCCESS The optional output value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS linear solver has not been initialized.

 $F2003\ \mathrm{Name}\ FCVodeGetLinSolvStats$

${\tt CVodeGetLastLinFlag}$

Call flag = CVodeGetLastLinFlag(cvode_mem, &lsflag);

Description The function CVodeGetLastLinFlag returns the last return value from a CVLS routine.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

lsflag (long int) the value of the last return flag from a CVLS function.

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

CVLS_SUCCESS The optional output value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer is NULL.

CVLS_LMEM_NULL The CVLS linear solver has not been initialized.

Notes

If the CVLS setup function failed (i.e., CVode returned CV_LSETUP_FAIL) when using the SUNLINSOL_DENSE or SUNLINSOL_BAND modules, then the value of lsflag is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the (dense or banded) Jacobian matrix.

If the CVLS 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 CVLS solve function failed (i.e., CVode returned CV_LSOLVE_FAIL), then lsflag contains the error return flag from the SUNLINSOL object, which will be one of: SUNLS_MEM_NULL, indicating that the SUNLINSOL memory is NULL;

SUNLS_ATIMES_FAIL_UNREC, indicating an unrecoverable failure in the Jv function; SUNLS_PSOLVE_FAIL_UNREC, indicating that the preconditioner solve function psolve failed unrecoverably; SUNLS_GS_FAIL, indicating a failure in the Gram-Schmidt procedure (SPGMR and SPFGMR only); SUNLS_QRSOL_FAIL, indicating that the matrix R was found to be singular during the QR solve phase (SPGMR and SPFGMR only); or SUNLS_PACKAGE_FAIL_UNREC, indicating an unrecoverable failure in an external iterative linear solver package.

The previous routines CVDlsGetLastFlag and CVSpilsGetLastFlag 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 FCVodeGetLastLinFlag

CVodeGetLinReturnFlagName

Call name = CVodeGetLinReturnFlagName(lsflag);

Description The function CVodeGetLinReturnFlagName returns the name of the CVLS constant corresponding to lsflag.

Arguments The only argument, of type long int, is a return flag from a CVLS 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 routine returns "NONE".

Notes

The previous routines CVDlsGetReturnFlagName and CVSpilsGetReturnFlagName 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 FCVodeGetLinReturnFlagName

4.5.10.6 Diagonal linear solver interface optional output functions

The following optional outputs are available from the CVDIAG module: workspace requirements, number of calls to the right-hand side routine for finite-difference Jacobian approximation, and last return value from a CVDIAG 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 here (e.g. lenrwLS).

CVDiagGetWorkSpace

Call flag = CVDiagGetWorkSpace(cvode_mem, &lenrwLS, &leniwLS);

Description The function CVDiagGetWorkSpace returns the CVDIAG real and integer workspace sizes.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

lenrwLS (long int) the number of realtype values in the CVDIAG workspace.
leniwLS (long int) the number of integer values in the CVDIAG workspace.

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

CVDIAG_SUCCESS The optional output value have been successfully set.

CVDIAG_MEM_NULL The cvode_mem pointer is NULL.

CVDIAG_LMEM_NULL The CVDIAG linear solver has not been initialized.

Notes In terms of the problem size N, the actual size of the real workspace is roughly 3N

realtype words.

F2003 Name FCVDiagGetWorkSpace

CVDiagGetNumRhsEvals

Call flag = CVDiagGetNumRhsEvals(cvode_mem, &nfevalsLS);

Description The function CVDiagGetNumRhsEvals returns the number of calls made to the user-

supplied right-hand side function due to the finite difference Jacobian approximation.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

nfevalsLS (long int) the number of calls made to the user-supplied right-hand side

function.

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

CVDIAG_SUCCESS The optional output value has been successfully set.

 ${\tt CVDIAG_MEM_NULL} \quad {\tt The \ cvode_mem \ pointer \ is \ NULL}.$

CVDIAG_LMEM_NULL The CVDIAG linear solver has not been initialized.

Notes The number of diagonal approximate Jacobians formed is equal to the number of calls

made to the linear solver setup function (see CVodeGetNumLinSolvSetups).

 $F2003 \; \mathrm{Name} \; \; \mathsf{FCVDiagGetNumRhsEvals}$

CVDiagGetLastFlag

Call flag = CVDiagGetLastFlag(cvode_mem, &lsflag);

Description The function CVDiagGetLastFlag returns the last return value from a CVDIAG routine.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

1sflag (long int) the value of the last return flag from a CVDIAG function.

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

CVDIAG_SUCCESS The optional output value has been successfully set.

CVDIAG_MEM_NULL The cvode_mem pointer is NULL.

CVDIAG_LMEM_NULL The CVDIAG linear solver has not been initialized.

Notes If the CVDIAG setup function failed (CVode returned CV_LSETUP_FAIL), the value of

lsflag is equal to CVDIAG_INV_FAIL, indicating that a diagonal element with value zero was encountered. The same value is also returned if the CVDIAG solve function failed

(CVode returned CV_LSOLVE_FAIL).

F2003 Name FCVDiagGetLastFlag

CVDiagGetReturnFlagName

Call name = CVDiagGetReturnFlagName(lsflag);

Description The function CVDiagGetReturnFlagName returns the name of the CVDIAG constant

corresponding to lsflag.

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

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

F2003 Name FCVDiagGetReturnFlagName

4.5.11 CVODE reinitialization function

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

The use of CVodeReInit requires that the maximum method order, denoted by maxord, be no larger for the new problem than for the previous problem. This condition is automatically fulfilled if the multistep method parameter 1mm is unchanged (or changed from CV_ADAMS to CV_BDF) and the default value for maxord is specified.

If there are changes to the linear solver specifications, make the appropriate calls to either the linear solver objects themselves, or to the CVLS interface routines, as described in §4.5.3. Otherwise, all solver inputs set previously remain in effect.

One important use of the CVodeReInit function is in the treating of jump discontinuities in the RHS 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 ODE model, using a call to CVodeReInit. 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 RHS 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 RHS 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.

CVodeReInit

Call flag = CVodeReInit(cvode_mem, t0, y0);

Description The function CVodeReInit provides required problem specifications and reinitializes

CVODE.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

t0 (realtype) is the initial value of t. y0 (N_Vector) is the initial value of y.

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

CV_SUCCESS The call to CVodeReInit was successful.

CV_MEM_NULL The CVODE memory block was not initialized through a previous call to CVodeCreate.

CV_NO_MALLOC Memory space for the CVODE memory block was not allocated through a previous call to CVodeInit.

CV_ILL_INPUT An input argument to CVodeReInit has an illegal value.

Notes If an error occurred, CVodeReInit also sends an error message to the error handler

function.

F2003 Name FCVodeReInit

4.6 User-supplied functions

The user-supplied functions consist of one function defining the ODE, (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 iterative algorithms.

4.6.1 ODE right-hand side

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

CVRhsFn

 $\label{eq:local-type} Definition \qquad \text{typedef int (*CVRhsFn)(real-type t, N_Vector y, N_Vector ydot,}$

void *user_data);

Purpose This function computes the ODE right-hand side for a given value of the independent

variable t and state vector y.

Arguments t is the current value of the independent variable.

y is the current value of the dependent variable vector, y(t).

ydot is the output vector f(t, y).

user_data is the user_data pointer passed to CVodeSetUserData.

Return value A CVRhsFn should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if it failed unrecov-

erably (in which case the integration is halted and CV_RHSFUNC_FAIL is returned).

Notes Allocation of memory for ydot is handled within CVODE.

A recoverable failure error return from the CVRhsFn is typically used to flag a value of the dependent variable y that is "illegal" in some way (e.g., negative where only a nonnegative value is physically meaningful). If such a return is made, CVODE 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 right-hand side 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 right-hand side function is called the first time during the following integration step, but a successful step cannot be undone.)

There are two other situations in which recovery is not possible even if the right-hand side function returns a recoverable error flag. One is when this occurs at the very first call to the CVRhsFn (in which case CVODE returns CV_FIRST_RHSFUNC_ERR). The other is when a recoverable error is reported by CVRhsFn after an error test failure, while the linear multistep method order is equal to 1 (in which case CVODE returns CV_UNREC_RHSFUNC_ERR).

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 CVodeSetErrFile), the user may provide a function of type CVErrHandlerFn to process any such messages. The function type CVErrHandlerFn is defined as follows:

CVErrHandlerFn

Definition typedef void (*CVErrHandlerFn)(int error_code, const char *module, const char *function, char *msg,

void *eh_data);

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

Arguments error_code is the error code.

module is the name of the CVODE 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

CVodeSetErrHandlerFn.

Return value A CVErrHandlerFn function has no return value.

Notes error_code is negative for errors and positive (CV_WARNING) for warnings. If a function

that returns a pointer to memory encounters an error, it sets error_code to 0.

4.6.3 Monitor function

A user may provide a function of type CVMonitorFn to monitor the integrator progress throughout a simulation. For example, a user may want to check integrator statistics as a simulation progresses.

CVMonitorFn

Definition typedef void (*CVMonitorFn)(void* cvode_mem, void* user_data);

Purpose This function is used to monitor the CVODE integrator throughout a simulation.

Arguments cvode_mem the CVODE memory block

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

CVodeSetUserData.

Return value A CVMonitorFn should return 0 if successful, or a negative value if unsuccessful.

Notes This function should only be utilized for monitoring the integrator.

4.6.4 Error weight function

As an alternative to providing the relative and absolute tolerances, the user may provide a function of type CVEwtFn to compute a vector ewt containing the weights in the WRMS norm $||v||_{WRMS} = \sqrt{(1/N)\sum_{i=1}^{N}(W_i\cdot v_i)^2}$. These weights will be used in place of those defined by Eq. (2.8). The function type CVEwtFn is defined as follows:

CVEwtFn

Definition typedef int (*CVEwtFn) (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

CVodeSetUserData.

Return value A CVEwtFn function type must return 0 if it successfully set the error weights and -1

otherwise.

Notes Allocation of memory for ewt is handled within CVODE.

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.





4.6.5 Rootfinding function

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

CVRootFn

Definition typedef int (*CVRootFn)(realtype t, N_Vector y, realtype *gout, void *user_data);

Purpose This function implements a vector-valued function g(t, y) such that the roots of the nrtfn components $q_i(t, y)$ are sought.

Arguments t is the current value of the independent variable.

y is the current value of the dependent variable vector, y(t). gout is the output array, of length nrtfn, with components $q_i(t, y)$.

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

Return value A CVRootFn should return 0 if successful or a non-zero value if an error occurred (in which case the integration is halted and CVode returns CV_RTFUNC_FAIL).

Notes Allocation of memory for gout is automatically handled within CVODE.

4.6.6 Projection function

When solving an IVP with a constraint equation and providing a user-defined projection operation the projection function must have type CVProjFn, defined as follows:

CVProjFn

Notes

Definition typedef int (*CVProjFn)(realtype t, N_Vector your, N_Vector corr, realtype epsProj, N_Vector err, void *user_data);

Purpose This function computes the projection of the solution and, if enabled, the error on to the constraint manifold.

Arguments t is the current value of the independent variable.

yeur is the current value of the dependent variable vector, y(t).

corr is the correction, c, to the dependent variable vector so that y(t) + c satisfies

the constraint equation.

epsProj is the tolerance to use in the nonlinear solver stopping test when solving the

nonlinear constrainted least squares problem.

err is on input the current error estimate, if error projection is enabled (the

default) then this should be overwritten with the projected error on output.

If error projection is disabled then err is NULL.

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

Return value A CVProjFn should return 0 if successful, a negative value if an unrecoverable error occurred (the integration is halted), or a positive value if a recoverable error occurred (the integrator will, in most cases, try to correct and reattempt the step).

The tolerance passed to the projection function (epsProj) is the tolerance on the iteration update in the WRMS norm, i.e., the solve should stop when the WRMS norm of the current iterate update is less than epsProj.

If needed by the user's projection routine, the error weight vector can be accessed by calling CVodeGetErrWeights, and the unit roundoff is available as UNIT_ROUNDOFF defined in sundials_types.h.

4.6.7Jacobian construction (matrix-based linear solvers)

If a matrix-based linear solver module is used (i.e., a non-NULL SUNMATRIX object was supplied to CVodeSetLinearSolver), the user may optionally provide a function of type CVLsJacFn for evaluating the Jacobian of the ODE right-hand side function (or an approximation of it). CVLsJacFn is defined as follows:

CVLsJacFn

typedef int (*CVLsJacFn)(realtype t, N_Vector y, N_Vector fy, Definition SUNMatrix Jac, void *user_data, N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);

Purpose This function computes the Jacobian matrix $J = \partial f/\partial y$ (or an approximation to it).

Arguments is the current value of the independent variable. t

> is the current value of the dependent variable vector, namely the predicted У

value of y(t).

is the current value of the vector f(t, y). fy

Jac is the output Jacobian matrix (of type SUNMatrix).

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

tmp1

tmp2

tmp3 are pointers to memory allocated for variables of type N_Vector which can be used by a CVLsJacFn function as temporary storage or work space.

Return value A CVLsJacFn should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct, while CVLS sets last_flag to CVLS_JACFUNC_RECVR), or a negative value if it failed unrecoverably (in which case the integration is halted, CVode returns CV_LSETUP_FAIL and CVLS sets last_flag to CVLS_JACFUNC_UNRECVR).

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 CVLsJacFn function is preceded by a call to the CVRhsFn user function with the same (t,y) arguments. Thus, the Jacobian function can use any auxiliary data that is computed and saved during the evaluation of the ODE right-hand side. In the case of a user-supplied or external nonlinear solver, this is also true if the nonlinear system function is evaluated prior to calling the linear solver setup function (see §10.1.4 for more information).

If the user's CVLsJacFn function uses difference quotient approximations, then it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to cv_mem to user_data and then use the CVodeGet* 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 N by N dense matrix Jac with an approximation to the Jacobian matrix J(t,y) at the point (t, y). 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 SUN-MATRIX_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 N by N banded matrix Jac with the elements of the Jacobian J(t,y) at the point (t,y). The accessor macros SM_ELEMENT_B, SM_COLUMN_B, and SM_COLUMN_ELEMENT_B allow the user to read and write band 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 band matrix Jac, counting from 0. This macro is meant for use in small problems for which efficiency of access is not a major concern. Thus, in terms of the indices m and n ranging from 1 to N with (m,n) within the band defined by mupper and mlower, the Jacobian element $J_{m,n}$ can be loaded using the statement SM_ELEMENT_B(J, m-1, n-1) = $J_{m,n}$. The elements within the band are those with -mupper \leq m-n \leq mlower. Alternatively, SM_COLUMN_B(J, j) returns a pointer to the diagonal element of the j-th column of Jac, and if we assign this address to realtype *col_j, then the i-th element of the j-th column is given by SM_COLUMN_ELEMENT_B(col_j, i, j), counting from 0. Thus, for (m,n) within the band, $J_{m,n}$ can be loaded by setting col_n = SM_COLUMN_B(J, n-1); 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 SUN-MATRIX_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 $\S 8.4$.

sparse:

A user-supplied sparse Jacobian function must load the N by N compressed-sparse-column or compressed-sparse-row matrix Jac with an approximation to the Jacobian matrix J(t,y) at the point (t,y). 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 CVDlsJacFn is identical to CVLsJacFn, 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 Linear system construction (matrix-based linear solvers)

With matrix-based linear solver modules, as an alternative to optionally supplying a function for evaluating the Jacobian of the ODE right-hand side function, the user may optionally supply a function

of type CVLsLinSysFn for evaluating the linear system, $M = I - \gamma J$ (or an approximation of it). CVLsLinSysFn is defined as follows:

```
CVLsLinSysFn
```

Definition typedef int (*CVLsLinSysFn)(realtype t, N_Vector y, N_Vector fy, SUNMatrix M, booleantype jok, booleantype *jcur, realtype gamma, void *user_data, N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);

Purpose This function computes the linear system matrix $M = I - \gamma J$ (or an approximation to it).

Arguments t is the current value of the independent variable.

y is the current value of the dependent variable vector, namely the predicted value of y(t).

fy is the current value of the vector f(t, y).

M is the output linear system matrix (of type SUNMatrix).

jok is an input flag indicating whether the Jacobian-related data needs to be updated. The jok flag enables reusing of Jacobian data across linear solves however, the user is responsible for storing Jacobian data for reuse. jok = SUNFALSE means that the Jacobian-related data must be recomputed from scratch. jok = SUNTRUE means that the Jacobian data, if saved from the previous call to this function, can be reused (with the current value of gamma). A call with jok = SUNTRUE can only occur after a call with jok = SUNTRUE can only occur after a call with jok =

SUNFALSE.

jcur is a pointer to a flag which should be set to SUNTRUE if Jacobian data was recomputed, or set to SUNFALSE if Jacobian data was not recomputed, but saved data was still reused.

gamma is the scalar γ appearing in the matrix $M = I - \gamma J$.

user_data is a pointer to user data, the same as the user_data parameter passed to the function CVodeSetUserData.

tmp1

tmp3 are pointers to memory allocated for variables of type N_Vector which can be used by a CVLsLinSysFn function as temporary storage or work space.

Return value A CVLsLinSysFn should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct, while CVLS sets last_flag to CVLS_JACFUNC_RECVR), or a negative value if it failed unrecoverably (in which case the integration is halted, CVode returns CV_LSETUP_FAIL and CVLS sets last_flag to CVLS_JACFUNC_UNRECVR).

4.6.9 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 CVodeSetLinearSolver), the user may provide a function of type CVLsJacTimesVecFn 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.

CVLsJacTimesVecFn

```
Definition typedef int (*CVLsJacTimesVecFn)(N_Vector v, N_Vector Jv, realtype t, N_Vector y, N_Vector fy, void *user_data, N_Vector tmp);
```

Purpose This function computes the product $Jv = (\partial f/\partial y)v$ (or an approximation to it).

Arguments v is the vector by which the Jacobian must be multiplied.

Jv is the output vector computed.

t is the current value of the independent variable.

y is the current value of the dependent variable vector.

fy is the current value of the vector f(t, y).

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

CVodeSetUserData.

tmp is a pointer to memory allocated for a variable of type N_Vector which can

be used for work space.

Return value The value returned by the Jacobian-vector product function should be 0 if successful.

Any other return value will result in an unrecoverable error of the generic Krylov solver,

in which case the integration is halted.

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).

If the user's CVLsJacTimesVecFn function uses difference quotient approximations, it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to cv_mem to user_data and then use the CVodeGet* 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 CVSpilsJacTimesVecFn is identical to CVLsJacTimesVecFn, 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.

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

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

${\tt CVLsJacTimesSetupFn}$

Definition typedef int (*CVLsJacTimesSetupFn) (realtype t, N_Vector y,

 $N_{ ext{-}}$ Vector fy, void *user_data);

Purpose This function preprocesses and/or evaluates Jacobian-related data needed by the Jacobian-

times-vector routine.

Arguments t is the current value of the independent variable.

y is the current value of the dependent variable vector.

fy is the current value of the vector f(t, y).

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

CVodeSetUserData.

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 CVRhsFn user function with the same (t,y) arguments. Thus, the setup function can use any

auxiliary data that is computed and saved during the evaluation of the ODE right-hand

side.

If the user's CVLsJacTimesSetupFn function uses difference quotient approximations, it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to cv_mem to user_data and then use the CVodeGet* 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 CVSpilsJacTimesSetupFn is identical to CVLsJacTimesSetupFn, 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.

4.6.11 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 may be either a left or right preconditioner matrix. Here P should approximate (at least crudely) the matrix $M=I-\gamma J$, where $J=\partial f/\partial y$. If preconditioning is done on both sides, the product of the two preconditioner matrices should approximate M. This function must be of type CVLsPrecSolveFn, defined as follows:

CVLsPrecSolveFn Definition typedef int (*CVLsPrecSolveFn)(realtype t, N_Vector y, N_Vector fy, N_Vector r, N_Vector z, realtype gamma, realtype delta, int lr, void *user_data); Purpose This function solves the preconditioned system Pz = r. is the current value of the independent variable. Arguments is the current value of the dependent variable vector. У is the current value of the vector f(t, y). fy is the right-hand side vector of the linear system. r is the computed output vector. is the scalar γ appearing in the matrix given by $M = I - \gamma J$. gamma 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 the weighted l_2 norm, i.e., $\sqrt{\sum_i (Res_i \cdot ewt_i)^2} <$ delta. To obtain the N_Vector ewt, call CVodeGetErrWeights (see §4.5.10.2). lr is an input flag indicating whether the preconditioner solve function is to use the left preconditioner (lr = 1) or the right preconditioner (lr = 2); user_data is a pointer to user data, the same as the user_data parameter passed to the function CVodeSetUserData.

Return value The value 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), or negative for an unrecoverable error (in which case the integration is halted).

The previous function type CVSpilsPrecSolveFn is identical to CVLsPrecSolveFn, 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.

4.6.12 Preconditioner setup (iterative linear solvers)

Notes

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

CVLsPrecSetupFn

Definition typedef int (*CVLsPrecSetupFn) (realtype t, N_Vector y, N_Vector fy,

> booleantype jok, booleantype *jcurPtr, realtype gamma, void *user_data);

Purpose This function preprocesses and/or evaluates Jacobian-related data needed by the preconditioner.

is the current value of the independent variable.

is the current value of the dependent variable vector, namely the predicted value of y(t).

fy is the current value of the vector f(t, y).

is an input flag indicating whether the Jacobian-related data needs to be updated. The jok argument provides for the reuse of Jacobian data in the preconditioner solve function. jok = SUNFALSE means that the Jacobianrelated data must be recomputed from scratch. jok = SUNTRUE means that the Jacobian data, if saved from the previous call to this function, can be reused (with the current value of gamma). A call with jok = SUNTRUE can

only occur after a call with jok = SUNFALSE.

is a pointer to a flag which should be set to SUNTRUE if Jacobian data was jcurPtr recomputed, or set to SUNFALSE if Jacobian data was not recomputed, but

saved data was still reused.

is the scalar γ appearing in the matrix $M = I - \gamma J$. gamma

user_data is a pointer to user data, the same as the user_data parameter passed to the function CVodeSetUserData.

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), or negative for an unrecoverable error (in which case the integration is halted).

> The operations performed by this function might include forming a crude approximate Jacobian and performing an LU factorization of the resulting approximation to M= $I - \gamma J$.

> With the default nonlinear solver (the native SUNDIALS Netwon method), each call to the preconditioner setup function is preceded by a call to the CVRhsFn user function with the same (t,y) arguments. Thus, the preconditioner setup function can use any auxiliary data that is computed and saved during the evaluation of the ODE right-hand side. In the case of a user-supplied or external nonlinear solver, this is also true if the nonlinear system 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 CVLsPrecSetupFn function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to cv_mem to user_data and then use the CVodeGet* 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 CVSpilsPrecSetupFn is identical to CVLsPrecSetupFn, 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.

Arguments

jok

Notes

4.7 Preconditioner modules

The efficiency of Krylov iterative methods for the solution of linear systems can be greatly enhanced through preconditioning. For problems in which the user cannot define a more effective, problem-specific preconditioner, CVODE provides a banded preconditioner in the module CVBANDPRE and a band-block-diagonal preconditioner module CVBBDPRE.

4.7.1 A serial banded preconditioner module

This preconditioner provides a band matrix preconditioner for use with iterative SUNLINSOL modules through the CVLS linear solver interface, in a serial setting. It uses difference quotients of the ODE right-hand side function f to generate a band matrix of bandwidth $m_l + m_u + 1$, where the number of super-diagonals (m_u , the upper half-bandwidth) and sub-diagonals (m_l , the lower half-bandwidth) are specified by the user, and uses this to form a preconditioner for use with the Krylov linear solver. Although this matrix is intended to approximate the Jacobian $\partial f/\partial y$, it may be a very crude approximation. The true Jacobian need not be banded, or its true bandwidth may be larger than $m_l + m_u + 1$, as long as the banded approximation generated here is sufficiently accurate to speed convergence as a preconditioner.

In order to use the CVBANDPRE module, the user need not define any additional functions. Aside from the header files required for the integration of the ODE problem (see §4.3), to use the CVBANDPRE module, the main program must include the header file cvode_bandpre.h which declares the needed function prototypes. The following is a summary of the usage of this module. Steps that are unchanged from the skeleton program presented in §4.4 are grayed out.

- 1. Initialize multi-threaded environment, if appropriate
- 2. Set problem dimensions etc.
- 3. Set vector of initial values
- 4. Create CVODE object
- 5. Initialize CVODE solver
- 6. Specify integration tolerances

7. Create linear solver object

When creating the iterative linear solver object, specify the type of preconditioning (PREC_LEFT or PREC_RIGHT) to use.

- 8. Set linear solver optional inputs
- 9. Attach linear solver module

10. Initialize the CVBANDPRE preconditioner module

Specify the upper and lower half-bandwidths (mu and ml, respectively) and call

flag = CVBandPrecInit(cvode_mem, N, mu, ml);

to allocate memory and initialize the internal preconditioner data.

11. Set optional inputs

Note that the user should not overwrite the preconditioner setup function or solve function through calls to the CVodeSetPreconditioner optional input function.

- 12. Create nonlinear solver object
- 13. Attach nonlinear solver module

- 14. Set nonlinear solver optional inputs
- 15. Specify rootfinding problem
- 16. Advance solution in time

17. Get optional outputs

Additional optional outputs associated with CVBANDPRE are available by way of two routines described below, CVBandPrecGetWorkSpace and CVBandPrecGetNumRhsEvals.

- 18. Deallocate memory for solution vector
- 19. Free solver memory
- 20. Free nonlinear solver memory
- 21. Free linear solver memory

The CVBANDPRE preconditioner module is initialized and attached by calling the following function:

CVBandPrecInit

Call flag = CVBandPrecInit(cvode_mem, N, mu, ml);

Description The function CVBandPrecInit initializes the CVBANDPRE preconditioner and allocates required (internal) memory for it.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

N (sunindextype) problem dimension.

mu (sunindextype) upper half-bandwidth of the Jacobian approximation.

ml (sunindextype) lower half-bandwidth of the Jacobian approximation.

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

CVLS_SUCCESS The call to CVBandPrecInit was successful.

CVLS_MEM_NULL The cvode_mem pointer was NULL.

CVLS_MEM_FAIL A memory allocation request has failed.

CVLS_LMEM_NULL A CVLS linear solver memory was not attached.

CVLS_ILL_INPUT The supplied vector implementation was not compatible with block band preconditioner.

Notes The banded approximate Jacobian will have nonzero elements only in locations (i, j) with $-ml \le j - i \le mu$.

F2003 Name FCVBandPrecInit

The following three optional output functions are available for use with the CVBANDPRE module:

CVBandPrecGetWorkSpace

Call flag = CVBandPrecGetWorkSpace(cvode_mem, &lenrwBP, &leniwBP);

Description The function CVBandPrecGetWorkSpace returns the sizes of the CVBANDPRE real and integer workspaces.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

lenrwBP (long int) the number of realtype values in the CVBANDPRE workspace.
leniwBP (long int) the number of integer values in the CVBANDPRE workspace.

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

CVLS_SUCCESS The optional output values have been successfully set.

CVLS_PMEM_NULL The CVBANDPRE preconditioner has not been initialized.

Notes

The workspace requirements reported by this routine correspond only to memory allocated within the CVBANDPRE module (the banded matrix approximation, banded SUNLINSOL object, and temporary vectors).

The workspaces referred to here exist in addition to those given by the corresponding function CVodeGetLinWorkSpace.

F2003 Name FCVBandPrecGetWorkSpace

CVBandPrecGetNumRhsEvals

Call flag = CVBandPrecGetNumRhsEvals(cvode_mem, &nfevalsBP);

Description The function CVBandPrecGetNumRhsEvals returns the number of calls made to the

user-supplied right-hand side function for the finite difference banded Jacobian approx-

imation used within the preconditioner setup function.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

nfevalsBP (long int) the number of calls to the user right-hand side function.

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

CVLS_SUCCESS The optional output value has been successfully set.

CVLS_PMEM_NULL The CVBANDPRE preconditioner has not been initialized.

Notes The counter nfevalsBP is distinct from the counter nfevalsLS returned by the corre-

sponding function CVodeGetNumLinRhsEvals and nfevals returned by

CVodeGetNumRhsEvals. The total number of right-hand side function evaluations is the

sum of all three of these counters.

F2003 Name FCVBandPrecGetNumRhsEvals

4.7.2 A parallel band-block-diagonal preconditioner module

A principal reason for using a parallel ODE solver such as CVODE 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 (2.6) 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 [39] and is included in a software module within the CVODE package. This module works with the parallel vector module NVECTOR_PARALLEL and is usable with any of the Krylov iterative linear solvers through the CVLs interface. It 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 CVBBDPRE.

One way to envision these preconditioners is to think of the domain of the computational PDE problem as being subdivided into M non-overlapping subdomains. Each of these subdomains is then assigned to one of the M processes to be used to solve the ODE system. The basic idea is to isolate the preconditioning so that it is local to each process, and also to use a (possibly cheaper) approximate right-hand side function. This requires the definition of a new function g(t,y) which approximates the function f(t,y) in the definition of the ODE system (2.1). However, the user may set g = f. Corresponding to the domain decomposition, there is a decomposition of the solution vector y into M disjoint blocks y_m , and a decomposition of g into blocks g_m . The block g_m depends both on y_m

and on components of blocks $y_{m'}$ associated with neighboring subdomains (so-called ghost-cell data). Let \bar{y}_m denote y_m augmented with those other components on which g_m depends. Then we have

$$g(t,y) = [g_1(t,\bar{y}_1), g_2(t,\bar{y}_2), \dots, g_M(t,\bar{y}_M)]^T$$
(4.1)

and each of the blocks $g_m(t, \bar{y}_m)$ is uncoupled from the others.

The preconditioner associated with this decomposition has the form

$$P = \operatorname{diag}[P_1, P_2, \dots, P_M] \tag{4.2}$$

where

$$P_m \approx I - \gamma J_m \tag{4.3}$$

and J_m is a difference quotient approximation to $\partial g_m/\partial y_m$. This matrix is taken to be banded, with upper and lower half-bandwidths mudq and mldq defined as the number of non-zero diagonals above and below the main diagonal, respectively. The difference quotient approximation is computed using mudq + mldq +2 evaluations of g_m , but only a matrix of bandwidth mukeep + mlkeep +1 is retained. Neither pair of parameters need be the true half-bandwidths of the Jacobian of the local block of g, if smaller values provide a more efficient preconditioner. 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 treatments of the blocks P_m . For example, incomplete LU factorization or an iterative method could be used instead of banded LU factorization.

The CVBBDPRE module calls two user-provided functions to construct P: a required function ${\tt gloc}$ (of type CVLocalFn) which approximates the right-hand side function $g(t,y)\approx f(t,y)$ and which is computed locally, and an optional function ${\tt cfn}$ (of type CVCommFn) which performs all interprocess communication necessary to evaluate the approximate right-hand side g. These are in addition to the user-supplied right-hand side function ${\tt f}$. Both functions take as input the same pointer user_data that is passed by the user to CVodeSetUserData and that was passed to the user's function ${\tt f}$. The user is responsible for providing space (presumably within user_data) for components of ${\tt y}$ that are communicated between processes by ${\tt cfn}$, and that are then used by ${\tt gloc}$, which should not do any communication.

CVLocalFn

Notes

Definition typedef int (*CVLocalFn)(sunindextype Nlocal, realtype t, N_Vector y, N_Vector glocal, void *user_data);

Purpose This gloc function computes g(t,y). It loads the vector glocal as a function of t and

Arguments Nlocal is the local vector length.

t is the value of the independent variable.

y is the dependent variable. glocal is the output vector.

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

Return value A CVLocalFn should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and CVode returns CV_LSETUP_FAIL).

This function must assume that all interprocess communication of data needed to calculate glocal has already been done, and that this data is accessible within user_data.

The case where g is mathematically identical to f is allowed.

CVCommFn

Definition typedef int (*CVCommFn)(sunindextype Nlocal, realtype t, N_Vector y, void *user_data);

Purpose This cfn function performs all interprocess communication necessary for the execution

of the gloc function above, using the input vector y.

Arguments Nlocal is the local vector length.

t is the value of the independent variable.

y is the dependent variable.

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

Return value A CVCommFn should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and CVode returns CV_LSETUP_FAIL).

Notes The cfn function is expected to save communicated data in space defined within the data structure user_data.

Each call to the cfn function is preceded by a call to the right-hand side function f with the same (t, y) arguments. Thus, cfn can omit any communication done by f if relevant to the evaluation of glocal. If all necessary communication was done in f, then cfn = NULL can be passed in the call to CVBBDPrecInit (see below).

Besides the header files required for the integration of the ODE problem (see §4.3), to use the CVBBDPRE module, the main program must include the header file cvode_bbdpre.h which declares the needed function prototypes.

The following is a summary of the proper usage of this module. Steps that are unchanged from the skeleton program presented in $\S4.4$ are grayed out.

- 1. Initialize MPI environment
- 2. Set problem dimensions etc.
- 3. Set vector of initial values
- 4. Create CVODE object
- 5. Initialize CVODE solver
- 6. Specify integration tolerances

7. Create linear solver object

When creating the iterative linear solver object, specify the type of preconditioning (PREC_LEFT or PREC_RIGHT) to use.

- 8. Set linear solver optional inputs
- 9. Attach linear solver module

10. Initialize the CVBBDPRE preconditioner module

Specify the upper and lower half-bandwidths mudq and mldq, and mukeep and mlkeep, and call

to allocate memory and initialize the internal preconditioner data. The last two arguments of CVBBDPrecInit are the two user-supplied functions described above.

11. Set optional inputs

Note that the user should not overwrite the preconditioner setup function or solve function through calls to the CVodeSetPreconditioner optional input function.

- 12. Create nonlinear solver object
- 13. Attach nonlinear solver module
- 14. Set nonlinear solver optional inputs
- 15. Specify rootfinding problem
- 16. Advance solution in time

17. Get optional outputs

Additional optional outputs associated with CVBBDPRE are available by way of two routines described below, CVBBDPrecGetWorkSpace and CVBBDPrecGetNumGfnEvals.

- 18. Deallocate memory for solution vector
- 19. Free solver memory
- 20. Free nonlinear solver memory
- 21. Free linear solver memory

gloc

cfn

22. Finalize MPI

The user-callable functions that initialize (step 10 above) or re-initialize the CVBBDPRE preconditioner module are described next.

CVBBDPrecInit

CVBBDPrecI	nit	
Call	flag = CV	<pre>/BBDPrecInit(cvode_mem, local_N, mudq, mldq,</pre>
Description	The function CVBBDPrecInit initializes and allocates (internal) memory for the CVBB-DPRE preconditioner.	
Arguments	cvode_mem	(void *) pointer to the CVODE memory block.
	$local_N$	(sunindextype) local vector length.
	mudq	(sunindextype) upper half-bandwidth to be used in the difference quotient Jacobian approximation.
	mldq	$({\tt sunindextype})$ lower half-bandwidth to be used in the difference quotient Jacobian approximation.
	mukeep	$({\tt sunindextype})$ upper half-bandwidth of the retained banded approximate Jacobian block.
	mlkeep	$({\tt sunindextype})$ lower half-bandwidth of the retained banded approximate Jacobian block.
	dqrely	(realtype) the relative increment in components of y used in the difference quotient approximations. The default is $dqrely = \sqrt{unit\ roundoff}$, which

can be specified by passing dqrely = 0.0.

nication required for the computation of g(t, y).

(CVLocalFn) the C function which computes the approximation $g(t,y) \approx$

(CVCommFn) the optional C function which performs all interprocess commu-

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

f(t,y).

CVLS_SUCCESS The call to CVBBDPrecInit was successful.

CVLS_MEM_NULL The cvode_mem pointer was NULL.

CVLS_MEM_FAIL A memory allocation request has failed.

CVLS_LMEM_NULL A CVLS linear solver was not attached.

CVLS_ILL_INPUT The supplied vector implementation was not compatible with 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 local_N-1, it is replaced by 0 or local_N-1 accordingly.

The half-bandwidths mudq and mldq need not be the true half-bandwidths of the Jacobian of the local block of q 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 computational costs further.

For all four half-bandwidths, the values need not be the same on every processor.

F2003 Name FCVBBDPrecInit

The CVBBDPRE module also provides a reinitialization function to allow solving 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 CVodeReInit to re-initialize CVODE for a subsequent problem, a call to CVBBDPrecReInit 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 dqrely, or one of the user-supplied functions gloc and cfn. 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 CVLS "set" functions, must also be made (in the proper order).

CVBBDPrecReInit

Call flag = CVBBDPrecReInit(cvode_mem, mudq, mldq, dqrely);

Description The function CVBBDPrecReInit re-initializes the CVBBDPRE preconditioner.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

> (sunindextype) upper half-bandwidth to be used in the difference quotient mudq Jacobian approximation.

> (sunindextype) lower half-bandwidth to be used in the difference quotient mldq Jacobian approximation.

> (realtype) the relative increment in components of y used in the difference dgrely quotient approximations. The default is $dqrely = \sqrt{unit}$ roundoff, which can be specified by passing dqrely = 0.0.

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

CVLS_SUCCESS The call to CVBBDPrecReInit was successful.

CVLS_MEM_NULL The cvode_mem pointer was NULL.

CVLS_LMEM_NULL A CVLS linear solver memory was not attached.

CVLS_PMEM_NULL The function CVBBDPrecInit was not previously called.

Notes If one of the half-bandwidths mudq or mldq is negative or exceeds the value local_N-1, it is replaced by 0 or local_N-1 accordingly.

F2003 Name FCVBBDPrecReInit

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

CVBBDPrecGetWorkSpace

Call flag = CVBBDPrecGetWorkSpace(cvode_mem, &lenrwBBDP, &leniwBBDP);

Description The function CVBBDPrecGetWorkSpace returns the local CVBBDPRE real and integer

workspace sizes.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

 ${\tt lenrwBBDP} \ ({\tt long\ int}) \ local \ number \ of \ {\tt realtype} \ values \ in \ the \ {\tt CVBBDPRE} \ workspace.$

leniwBBDP (long int) local number of integer values in the CVBBDPRE workspace.

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

CVLS_SUCCESS The optional output value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer was NULL.

CVLS_PMEM_NULL The CVBBDPRE preconditioner has not been initialized.

Notes The workspace requirements reported by this routine correspond only to memory allo-

cated within the CVBBDPRE 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 CVodeGetLinWorkSpace.

F2003 Name FCVBBDPrecGetWorkSpace

CVBBDPrecGetNumGfnEvals

Call flag = CVBBDPrecGetNumGfnEvals(cvode_mem, &ngevalsBBDP);

Description The function CVBBDPrecGetNumGfnEvals returns the number of calls made to the user-

supplied gloc function due to the finite difference approximation of the Jacobian blocks

used within the preconditioner setup function.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

ngevalsBBDP (long int) the number of calls made to the user-supplied gloc function.

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

CVLS_SUCCESS The optional output value has been successfully set.

CVLS_MEM_NULL The cvode_mem pointer was NULL.

CVLS_PMEM_NULL The CVBBDPRE preconditioner has not been initialized.

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

In addition to the ngevalsBBDP gloc evaluations, the costs associated with CVBBDPRE also include nlinsetups LU factorizations, nlinsetups calls to cfn, npsolves banded backsolve calls, and nfevalsLS right-hand side function evaluations, where nlinsetups is an optional CVODE output and npsolves and nfevalsLS are linear solver optional outputs (see §4.5.10).

Chapter 5

Using CVODE for Fortran Applications

A Fortran 2003 module (fcvode_mod) as well as a Fortran 77 style interface (FCVODE) are provided to support the use of CVODE, for the solution of ODE systems dy/dt = f(t,y), in a mixed Fortran/C setting. While CVODE 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 CVODE Fortran 2003 Interface Module

The fcvode_mod Fortran module defines interfaces to most CVODE 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 CVODE function CVodeCreate is interfaced as FCVodeCreate. Thus, the steps to use CVODE 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 CVODE interface module can be accessed by the use statement, i.e. use fcvode_mod, and linking to the library libsundials_fcvode_mod.lib in addition to libsundials_cvode.lib.

The Fortran 2003 interface modules were generated with SWIG Fortran, a fork of SWIG [41]. 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.

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

Module Fortran 2003 Module Name			
NVECTOR	fsundials_nvector_mod		
NVECTOR_SERIAL	fnvector_serial_mod		
NVECTOR_PARALLEL	fnvector_parallel_mod		
NVECTOR_OPENMP	fnvector_openmp_mod		
NVECTOR_PTHREADS	fnvector_pthreads_mod		
NVECTOR_PARHYP	Not interfaced		
NVECTOR_PETSC	Not interfaced		
NVECTOR_CUDA	Not interfaced		
NVECTOR_RAJA	Not interfaced		
NVECTOR_MANYVECTOR	fnvector_manyvector_mod		
NVECTOR_MPIMANYVECTOR	$fnvector_mpimanyvector_mod$		
NVECTOR_MPIPLUSX	fnvector_mpiplusx_mod		
SUNMatrix	fsundials_matrix_mod		
SUNMATRIX_BAND	fsunmatrix_band_mod		
SUNMATRIX_DENSE	fsunmatrix_dense_mod		
SUNMATRIX_SPARSE	fsunmatrix_sparse_mod		
SUNLinearSolver	fsundials_linearsolver_mod		
SUNLINSOL_BAND	fsunlinsol_band_mod		
SUNLINSOL_DENSE	fsunlinsol_dense_mod		
SUNLINSOL_LAPACKBAND	Not interfaced		
SUNLINSOL_LAPACKDENSE	_		
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		

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.

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 Notable Fortran/C usage differences

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

5.1.3.1 Creating generic SUNDIALS objects

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

```
C code:
N_Vector x;
x = N_VNew_Serial(N);
Fortran code:
```

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

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

5.1.3.2 Arrays and pointers

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

```
C code:
N_Vector x
realtype* xdata;
long int leniw, lenrw;
x = N_VNew_Serial(N);
/* capturing a returned array/pointer */
xdata = N_VGetArrayPointer(x)
/* passing array/pointer to a function */
N_VSetArrayPointer(xdata, x)
/* pointers that are out-parameters */
N_VSpace(x, &leniw, &lenrw);
Fortran code:
type(N_Vector), pointer :: x
real(c_double), pointer :: xdataptr(:)
real(c_double)
                       :: xdata(N)
                        :: leniw(1), lenrw(1)
integer(c_long)
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.

C code:

```
SUNLinearSolver LS;
N_Vector x, b;
! SUNLinSolSolve expects a SUNMatrix or NULL
! as the second parameter.
ierr = SUNLinSolSolve(LS, NULL, x, b);
Fortran code:
type(SUNLinearSolver), pointer :: LS
type(SUNMatrix), pointer :: A
type(N_Vector), pointer :: x, b

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

5.1.3.5 Working with N_Vector arrays

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

example below demonstrates this:

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

5.1.3.6 Providing file pointers

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

```
FSUNDIALSFileOpen
Call
             fp = FSUNDIALSFileOpen(filename, mode)
             The function allocates a FILE * by calling the C function fopen.
Description
             filename (character(kind=C_CHAR, len=*)) - the path to the file to open
Arguments
                        (character(kind=C_CHAR, len=*)) - the mode string given to fopen It
             mode
                        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 sys-
                            tem/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 FCVODE, an Interface Module for FORTRAN Applications

The FCVODE interface module is a package of C functions which support the use of the CVODE. This package provides the necessary interface to CVODE for all supplied serial and parallel NVECTOR implementations.

5.2.1 Important note on portability

In this package, the names of the interface functions, and the names of the FORTRAN user routines called by them, appear as dummy names which are mapped to actual values by a series of definitions in the header files. By default, those mapping definitions depend in turn on the C macro 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).

5.2.2 Fortran Data Types

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

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

• int32_t - equivalent to an INTEGER or INTEGER*4 in FORTRAN

• 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

5.2.3 FCVODE routines

The user-callable functions, with the corresponding CVODE 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 CVODE module
 - FCVMALLOC interfaces to CVodeCreate, CVodeSetUserData, and CVodeInit, as well as one of CVodeSStolerances or CVodeSVtolerances.

- FCVREINIT interfaces to CVodeReInit.
- FCVSETIIN and FCVSETRIN interface to CVodeSet* functions.
- FCVEWTSET interfaces to CVodeWFtolerances.
- FCVODE interfaces to CVode, CVodeGet* functions, and to the optional output functions for the selected linear solver module.
- FCVDKY interfaces to the interpolated output function CVodeGetDky.
- FCVGETERRWEIGHTS interfaces to CVodeGetErrWeights.
- FCVGETESTLOCALERR interfaces to CVodeGetEstLocalErrors.
- FCVFREE interfaces to CVodeFree.
- Interface to the linear solver interfaces
 - FCVLSINIT interfaces to CVodeSetLinearSolver.
 - FCVLSSETEPSLIN interfaces to CVodeSetEpsLin.
 - FCVLSSETJAC interfaces to CVodeSetJacTimes.
 - FCVLSSETPREC interfaces to CVodeSetPreconditioner.
 - FCVDENSESETJAC interfaces to CVodeSetJacFn.
 - FCVBANDSETJAC interfaces to CVodeSetJacFn.
 - FCVSPARSESETJAC interfaces to CVodeSetJacFn.
 - FCVDIAG interfaces to CVDiag.
- Interface to the nonlinear solver interface
 - FCVNLSINIT interfaces to CVSetNonlinearSolver.

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

FCVODE routine	CVODE function	CVODE type of
(FORTRAN, user-supplied)	(C, interface)	interface function
FCVFUN	FCVf	CVRhsFn
FCVEWT	FCVEwtSet	CVEwtFn
FCVDJAC	FCVDenseJac	CVLsJacFn
FCVBJAC	FCVBandJac	CVLsJacFn
FCVSPJAC	FCVSparseJac	CVLsJacFn
FCVPSOL	FCVPSol	CVLsPrecSolveFn
FCVPSET	FCVPSet	CVLsPrecSetupFn
FCVJTIMES	FCVJtimes	CVLsJacTimesVecFn
FCVJTSETUP	FCVJTSetup	CVLsJacTimesSetupFn

In contrast to the case of direct use of CVODE, and of most FORTRAN ODE solvers, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program.

5.2.4 Usage of the FCVODE interface module

The usage of FCVODE 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 CVODE 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 FCVODE for rootfinding and with preconditioner modules is described in later subsections.

1. Right-hand side specification

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

```
SUBROUTINE FCVFUN(T, Y, YDOT, IPAR, RPAR, IER)
DIMENSION Y(*), YDOT(*), IPAR(*), RPAR(*)
```

It must set the YDOT array to f(t,y), the right-hand side of the ODE system, as function of T=t and the array Y=y. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted).

2. NVECTOR module initialization

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

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

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

3. SUNMATRIX module initialization

If using a nonlinear solver that requires a linear solver (e.g., the default Newton iteration) and the linear solver is a direct linear solver, then 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 nonlinear solver that requires a linear solver (e.g., the default Newton iteration) and one of the SUNLINSOL linear solver modules supplied with SUNDIALS, the user must make a call of the form

```
CALL FSUNBANDLINSOLINIT(...)

CALL FSUNCHUINIT(...)

CALL FSUNKLUINIT(...)

CALL FSUNLAPACKBANDINIT(...)

CALL FSUNLAPACKDENSEINIT(...)

CALL FSUNPCGINIT(...)

CALL FSUNSPBCGSINIT(...)

CALL FSUNSPFGMRINIT(...)

CALL FSUNSPFGMRINIT(...)

CALL FSUNSPTFQMRINIT(...)

CALL FSUNSPTFQMRINIT(...)
```

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 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. SUNNONLINSOL module initialization

By default CVODE uses the SUNNONLINSOL implementation of Newton's method defined by the SUNNONLINSOL_NEWTON module (see §10.3). To specify a non-default nonlinear solver in CVODE, the user's program must create a SUNNONLINSOL object by calling the appropriate FORTRAN interface function to the constructor routine (see Chapter 10). For example, to create the SUNNONLINSOL_FIXEDPOINT solver call the function

```
CALL FSUNFIXEDPOINTINIT(...)
```

in which the call sequence is described in §10.4.

6. Problem specification

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

FCVMALLOC

```
Call
               CALL FCVMALLOC(TO, YO, METH, IATOL, RTOL, ATOL,
                                 IOUT, ROUT, IPAR, RPAR, IER)
             This function provides required problem and solution specifications, specifies op-
Description
              tional inputs, allocates internal memory, and initializes CVODE.
                     is the initial value of t.
Arguments
             TΩ
              Y0
                     is an array of initial conditions.
             METH specifies the basic integration method: 1 for Adams (nonstiff) or 2 for BDF
              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 FCVEWTSET and provide the function FCVEWT.
                    is the relative tolerance (scalar).
              ATOL is the absolute tolerance (scalar or array).
                    is an integer array of length 21 for integer optional outputs.
                    is a real array of length 6 for real optional outputs.
             ROUT
              IPAR
                    is an integer array of user data which will be passed unmodified to all user-
                     provided routines.
              RPAR is a real array of user data which will be passed unmodified to all user-
                     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 CVODE integrator are listed in Table 5.4.

As an alternative to providing tolerances in the call to FCVMALLOC, 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 FCVEWT (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 FCVEWT 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 FCVMALLOC.

If the FCVEWT routine is provided, then, following the call to FCVMALOC, the user must make the call:

```
CALL FCVEWTSET (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.

7. Set optional inputs

Call FCVINSETIIN and/or FCVINSETRIN to set desired optional inputs, if any. See §5.2.5 for details.

8. Linear solver interface specification

To attach the linear solver (and optionally the matrix) objects initialized in steps 3 and 4 above, the user of FCVODE must initialize the CVLS linear solver interface. To attach any SUNLINSOL object (and optional SUNMATRIX object) to CVODE, then following calls to initialize the SUNLINSOL (and SUNMATRIX) object(s) in steps 3 and 4 above, the user must make the call:

```
CALL FCVLSINIT (IER)
```

IER is an error return flag set on 0 on success, -1 if a memory failure occurred, or -2 for an illegal input.

The previous routines FCVDLSINIT and FCVSPILSINIT 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.

CVLS with dense Jacobian matrix

As an option when using the CVLS interface with the SUNLINSOL_DENSE or SUNLINSOL_LAPACKDENSE linear solvers, the user may supply a routine that computes a dense approximation of the system Jacobian $J = \partial f/\partial y$. If supplied, it must have the following form:

```
SUBROUTINE FCVDJAC (NEQ, T, Y, FY, DJAC, H, IPAR, RPAR, & WK1, WK2, WK3, IER)
DIMENSION Y(*), FY(*), DJAC(NEQ,*), IPAR(*), RPAR(*), & WK1(*), WK2(*), WK3(*)
```

Typically this routine will use only NEQ, T, Y, and DJAC. It must compute the Jacobian and store it columnwise in DJAC. The input arguments T, Y, and FY contain the current values of t, y, and f(t,y), respectively. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. The vectors WK1, WK2, and WK3 of length NEQ are provided as work space for use in FCVDJAC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if FCVDJAC failed unrecoverably (in which case the integration is halted). NOTE: The argument NEQ has a type consistent with C type long int even in the case when the LAPACK dense solver is to be used.

If the user's FCVDJAC uses difference quotient approximations, it may need to use the error weight array EWT and current stepsize H in the calculation of suitable increments. The array EWT can be obtained by calling FCVGETERRWEIGHTS using one of the work arrays as temporary storage for EWT. 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 either RPAR or a common block.

If the FCVDJAC routine is provided, then, following the call to FCVLSINIT, the user must make the call:

```
CALL FCVDENSESETJAC (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.

CVLS with band Jacobian matrix

As an option when using the CVLS interface with the SUNLINSOL_BAND or SUNLINSOL_LAPACKBAND linear solvers, the user may supply a routine that computes a band approximation of the system Jacobian $J = \partial f/\partial y$. If supplied, it must have the following form:

```
SUBROUTINE FCVBJAC(NEQ, MU, ML, MDIM, T, Y, FY, BJAC, H, IPAR, RPAR, & WK1, WK2, WK3, IER)

DIMENSION Y(*), FY(*), BJAC(MDIM,*), IPAR(*), RPAR(*), 
& WK1(*), WK2(*), WK3(*)
```

Typically this routine will use only NEQ, MU, ML, T, Y, and BJAC. It must load the MDIM by N array BJAC with the Jacobian matrix at the current (t,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 input arguments T, Y, and FY contain the current values of t,y, and f(t,y), respectively. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. The vectors WK1, WK2, and WK3 of length NEQ are provided as work space for use in FCVBJAC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if FCVBJAC failed unrecoverably (in which case the integration is halted). NOTE: The arguments NEQ, MU, ML, and MDIM have a type consistent with C type long int even in the case when the LAPACK band solver is to be used.

If the user's FCVBJAC uses difference quotient approximations, it may need to use the error weight array EWT and current stepsize H in the calculation of suitable increments. The array EWT can be obtained by calling FCVGETERRWEIGHTS using one of the work arrays as temporary storage for EWT. 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 either RPAR or a common block.

If the FCVBJAC routine is provided, then, following the call to FCVLSINIT, the user must make the call:

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.

CVLS with sparse Jacobian matrix

When using the CVLS interface with the SUNLINSOL_KLU or SUNLINSOL_SUPERLUMT linear solvers, the user must supply the FCVSPJAC routine that computes a compressed-sparse-column or compressed-sparse-row approximation of the system Jacobian $J = \partial f/\partial y$. If supplied, it must have the following form:

```
SUBROUTINE FCVSPJAC(T, Y, FY, 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, JINDEXVALS and JINDEXPTRS, 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; Y, an array containing state variables; FY, an array containing state derivatives; 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 FCVMALLOC; RPAR, an array containing real user data that was passed to FCVMALLOC; 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 FCVSPJAC routine has been provided, then following the call to FCVLSINIT, the following call must be made

```
CALL FCVSPARSESETJAC (IER)
```

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

CVLS with Jacobian-vector product

As an option when using the CVLS linear solver interface, the user may supply a routine that computes the product of the system Jacobian $J = \partial f/\partial y$ and a given vector v. If supplied, it must have the following form:

```
SUBROUTINE FCVJTIMES (V, FJV, T, Y, FY, H, IPAR, RPAR, WORK, IER) DIMENSION V(*), FJV(*), Y(*), FY(*), IPAR(*), RPAR(*), WORK(*)
```

Typically this routine will use only T, Y, V, and FJV. It must compute the product vector Jv, where the vector v is stored in V, and store the product in FJV. The input arguments T, Y, and FY contain the current values of t, y, and f(t,y), respectively. On return, set IER = 0 if FCVJTIMES 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 FCVMALLOC. The vector WORK, of length commensurate with the input YO to FCVMALLOC, is provided as work space for use in FCVJTIMES.

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 FCVJTSETUP (T, Y, FY, H, IPAR, RPAR, IER) DIMENSION Y(*), FY(*), IPAR(*), RPAR(*)
```

Typically this routine will use only T and Y. It should compute any necessary data for subsequent calls to FCVJTIMES. On return, set IER = 0 if FCVJTSETUP 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 FCVMALLOC.

To indicate that the FCVJTIMES and FCVJTSETUP routines have been provided, then following the call to FCVLSINIT, the following call must be made

```
CALL FCVLSSETJAC (FLAG, IER)
```

with FLAG $\neq 0$ to specify use of the user-supplied Jacobian-times-vector setup and product routines. The argument IER is an error return flag which is 0 for success or non-zero if an error occurred.

The previous routine FCVSPILSETJAC 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 FCVLSSETJAC, the routine FCVJTSETUP must be provided, even if it is not needed, and it must return IER=0.

Notes

- (a) If the user's FCVJTIMES routine uses difference quotient approximations, it may need to use the error weight array EWT, the current stepsize H, and/or the unit roundoff, in the calculation of suitable increments.
- (b) If needed in FCVJTIMES or FCVJTSETUP, the error weight array EWT can be obtained by calling FCVGETERRWEIGHTS using a user-allocated array as temporary storage for EWT.
- (c) If needed in FCVJTIMES or FCVJTSETUP, the unit roundoff can be obtained as the optional output ROUT(6) (available after the call to FCVMALLOC) and can be passed using either the RPAR user data array, a common block or a module.

CVLS with preconditioning

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

```
SUBROUTINE FCVPSOL(T, Y, FY, R, Z, GAMMA, DELTA, LR, IPAR, RPAR, IER) DIMENSION Y(*), FY(*), R(*), Z(*), IPAR(*), RPAR(*)
```

It must solve the preconditioner linear system Pz=r, where r=R is input, and store the solution z in Z. Here P is the left preconditioner if LR=1 and the right preconditioner if LR=2. The preconditioner (or the product of the left and right preconditioners if both are nontrivial) should be an approximation to the matrix $I-\gamma J$, where I is the identity matrix, J is the system Jacobian, and $\gamma=\text{GAMMA}$. The input arguments T, Y, and FY contain the current values of t,y, and f(t,y), respectively. On return, set IER = 0 if FCVPSOL 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 FCVMALLOC.

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

<u>'!</u>

It must perform any evaluation of Jacobian-related data and preprocessing needed for the solution of the preconditioner linear systems by FCVPSOL. The input argument JOK allows for Jacobian data to be saved and reused: If JOK = 0, this data should be recomputed from scratch. If JOK = 1, a saved copy of it may be reused, and the preconditioner constructed from it. The input arguments T, Y, and FY contain the current values of t, y, and f(t,y), respectively. On return, set JCUR = 1 if Jacobian data was computed, and set JCUR = 0 otherwise. Also on return, set IER = 0 if FCVPSET 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 FCVMALLOC.

To indicate that the FCVPSET and FCVPSOL routines are supplied, then the user must call

CALL FCVLSSETPREC(FLAG, IER)

with FLAG $\neq 0$. The return flag IER is 0 if successful, or negative if a memory error occurred. In addition, the user program must include preconditioner routines FCVPSOL and FCVPSET.

The previous routine FCVSPILSETPREC 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 FCVLSSETPREC, the routine FCVPSET must be provided, even if it is not needed, and it must return IER=0.

Notes

- (a) If the user's FCVPSET routine uses difference quotient approximations, it may need to use the error weight array EWT, the current stepsize H, and/or the unit roundoff, in the calculation of suitable increments. Also, If FCVPSOL uses an iterative method in its solution, the residual vector $\rho = r Pz$ of the system should be made less than DELTA in weighted ℓ_2 norm, i.e. $\sqrt{\sum (\rho_i * \text{EWT}[i])^2} < \text{DELTA}$.
- (b) If needed in FCVPSOL or FCVPSET, the error weight array EWT can be obtained by calling FCVGETERRWEIGHTS using a user-allocated array as temporary storage for EWT.
- (c) If needed in FCVPSOL or FCVPSET, the unit roundoff can be obtained as the optional output ROUT(6) (available after the call to FCVMALLOC) and can be passed using either the RPAR user data array, a common block or a module.

CVDIAG diagonal linear solver interface

CVODE is also packaged with a CVODE-specific diagonal approximate Jacobian and linear solver interface. This choice is appropriate when the Jacobian can be well-approximated by a diagonal matrix. The user must make the call:

CALL FCVDIAG(IER)

IER is an error return flag set on 0 on success or -1 if a memory failure occurred.

There are no additional user-supplied routines for the CVDIAG interface.

Optional outputs specific to the CVDIAG case are listed in Table 5.4.

9. Nonlinear solver interface specification

If a non-default SUNNONLINSOL object was created in step 5, the user must attach it to CVODE with the call:

CALL FCVNLSINIT(IER)



IER is an error return flag set on 0 on success or -1 if an error occurred.

Once attached, the user may specify non-default inputs for the SUNNONLINSOL object (e.g. the maximum number of nonlinear iterations) by calling appropriate FORTRAN interface routines (see Chapter 10).

10. Problem solution

Carrying out the integration is accomplished by making calls as follows:

```
CALL FCVODE(TOUT, T, Y, 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 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 CVode 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).

11. Additional solution output

After a successful return from FCVODE, the routine FCVDKY may be used to obtain a derivative of the solution, of order up to the current method order, at any t within the last step taken. For this, make the following call:

```
CALL FCVDKY(T, K, DKY, IER)
```

where T is the value of t at which solution derivative is desired, and K is the derivative order $(0 \le K \le QU)$. On return, DKY is an array containing the computed K-th derivative of y. The value T must lie between TCUR - HU and TCUR. The return flag IER is set to 0 upon successful return or to a negative value to indicate an illegal input.

12. Problem reinitialization

To re-initialize the CVODE solver for the solution of a new problem of the same size as one already solved, make the following call:

```
CALL FCVREINIT(TO, YO, IATOL, RTOL, ATOL, IER)
```

The arguments have the same names and meanings as those of FCVMALLOC. FCVREINIT performs the same initializations as FCVMALLOC, but does no memory allocation, using instead the existing internal memory created by the previous FCVMALLOC call. The call to specify the linear system solution method may or may not be needed.

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 CVLS interface, as shown above. If only linear solver parameters are being modified, then these calls may be made without re-attaching to the CVLS interface.

13. Memory deallocation

To free the internal memory created by the call to FCVMALLOC, FCVLSINIT, FNVINIT* and FSUN***MATINIT, make the call

CALL FCVFREE

Integer optional inputs (FCVSETIIN)				
Key	Key Optional input			
MAX_ORD	Maximum LMM method order	5 (BDF), 12 (Adams)		
MAX_NSTEPS	MAX_NSTEPS Maximum no. of internal steps before t_{out}			
MAX_ERRFAIL Maximum no. of error test failures		7		
MAX_NITERS Maximum no. of nonlinear iterations		3		
MAX_CONVFAIL	Maximum no. of convergence failures	10		
HNIL_WARNS Maximum no. of warnings for $t_n + h = t_n$		10		
STAB_LIM	Flag to activate stability limit detection	0		

Table 5.3: Keys for setting FCVODE optional inputs

Real optional inputs (FCVSETRIN)

Key	Optional input	Default value
INIT_STEP	Initial step size	estimated
MAX_STEP	Maximum absolute step size	∞
MIN_STEP	Minimum absolute step size	0.0
STOP_TIME	Value of t_{stop}	undefined
NLCONV_COEF	Coefficient in the nonlinear convergence test	0.1

Real vector optional inputs (FCVSETVIN)

Key Optional Input		Default value
$CONSTR_VEC$	Inequality constraints on solution	undefined

5.2.5 FCVODE optional input and output

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

```
CALL FCVSETIIN(KEY, IVAL, IER)
CALL FCVSETRIN(KEY, RVAL, IER)
CALL FCVSETVIN(KEY, VVAL, IER)
```

where KEY is a quoted string indicating which optional input is set (see Table 5.3), IVAL is the integer input value to be used, RVAL is the real input value to be used, VVAL is the real input array to be used, and IER is an integer return flag which is set to 0 on success and a negative value if a failure occurred. The integer IVAL should be declared in a manner consistent with C type long int.

When using FCVSETVIN 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 CVodeSetConstraints (§4.5.8.1) for details.

The optional outputs from the CVODE 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 FCVMALLOC. Table 5.4 lists the entries in these two arrays and specifies the optional variable as well as the CVODE 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 FCVSET* calls and the optional outputs extracted from IOUT and ROUT, the following user-callable routines are available:

To obtain the error weight array EWT, containing the multiplicative error weights used the WRMS norms, make the following call:

```
CALL FCVGETERRWEIGHTS (EWT, IER)
```

Table 5.4: Description of the FCVODE optional output arrays ${\tt IOUT}$ and ${\tt ROUT}$

	_	- · · · · ·			
	Integer output array IOUT				
Index	Optional output	CVODE function			
CVODE main solver					
1	LENRW	CVodeGetWorkSpace			
2	LENIW	CVodeGetWorkSpace			
3	_				
4	NFE	CVodeGetNumRhsEvals			
5	NETF	CVodeGetNumErrTestFails			
6	NCFN	CVodeGetNumNonlinSolvConvFails			
7	NNI	CVodeGetNumNonlinSolvIters			
8	NSETUPS	CVodeGetNumLinSolvSetups			
9	QU	CVodeGetLastOrder			
10	QCUR	CVodeGetCurrentOrder			
11					
12	NGE	CVodeGetNumGEvals			
	CVLS line	ar solver interface			
13	LENRWLS	CVodeGetLinWorkSpace			
14	LENIWLS	CVodeGetLinWorkSpace			
15	LS_FLAG	CVodeGetLastLinFlag			
16	NFELS	CVodeGetNumLinRhsEvals			
17	NJE	CVodeGetNumJacEvals			
18	NJTS	CVodeGetNumJTSetupEvals			
19	NJTV	CVodeGetNumJtimesEvals			
20	NPE	CVodeGetNumPrecEvals			
21	NPS	CVodeGetNumPrecSolves			
22	NLI	CVodeGetNumLinIters			
23	NCFL	CVodeGetNumLinConvFails			
	CVDIAG linear solver interface				
13	LENRWLS	CVDiagGetWorkSpace			
14	LENIWLS	CVDiagGetWorkSpace			
15	LS_FLAG	CVDiagGetLastFlag			
1.0					

Real output array ROUT

 ${\tt CVDiagGetNumRhsEvals}$

16

NFELS

Index	Optional output	CVODE function			
1	1 HOU CVodeGetActualInitStep				
2	2 HU CVodeGetLastStep				
3	HCUR	CVodeGetCurrentStep			
4	TCUR	CVodeGetCurrentTime			
5 TOLSF CVodeGetTolScaleFactor		CVodeGetTolScaleFactor			
6 UROUND unit roundoff					

This computes the EWT array normally defined by Eq. (2.8). 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.

To obtain the estimated local errors, following a successful call to FCVSOLVE, make the following call:

```
CALL FCVGETESTLOCALERR (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.

5.2.6 Usage of the FCVROOT interface to rootfinding

The FCVROOT interface package allows programs written in FORTRAN to use the rootfinding feature of the CVODE solver module. The user-callable functions in FCVROOT, with the corresponding CVODE functions, are as follows:

- FCVROOTINIT interfaces to CVodeRootInit.
- FCVROOTINFO interfaces to CVodeGetRootInfo.
- FCVROOTFREE interfaces to CVodeRootFree.

Note that at this time, FCVROOT 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 FCVROTINFO).

In order to use the rootfinding feature of CVODE, the following call must be made, after calling FCVMALLOC but prior to calling FCVODE, to allocate and initialize memory for the FCVROOT module:

```
CALL FCVROOTINIT (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 CVODE memory was NULL, and -11 if a memory allocation failed. To specify the functions whose roots are to be found, the user must define the following routine:

```
SUBROUTINE FCVROOTFN (T, Y, G, IPAR, RPAR, IER)
DIMENSION Y(*), G(*), IPAR(*), RPAR(*)
```

It must set the G array, of length NRTFN, with components $g_i(t, y)$, as a function of T = t and the array Y = y. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. Set IER on 0 if successful, or on a non-zero value if an error occurred.

When making calls to FCVODE to solve the ODE 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 FCVROOTINFO (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 ± 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 FCVROOTFN, denoted NGE, can be obtained from IOUT(12). If the FCVODE/CVODE memory block is reinitialized to solve a different problem via a call to FCVREINIT, then the counter NGE is reset to zero.

To free the memory resources allocated by a prior call to FCVROOTINIT, make the following call:

```
CALL FCVROOTFREE
```

5.2.7 Usage of the FCVBP interface to CVBANDPRE

The FCVBP interface sub-module is a package of C functions which, as part of the FCVODE interface module, support the use of the CVODE solver with the serial NVECTOR_SERIAL module or multi-threaded NVECTOR_OPENMP or NVECTOR_PTHREADS, and the combination of the CVBANDPRE preconditioner module (see §4.7.1) with the CVLS interface and any of the Krylov iterative linear solvers.

The two user-callable functions in this package, with the corresponding CVODE function around which they wrap, are:

- FCVBPINIT interfaces to CVBandPrecInit.
- FCVBPOPT interfaces to CVBANDPRE optional output functions.

As with the rest of the FCVODE routines, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file fcvbp.h.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in §5.2.4 are grayed-out.

- 1. Right-hand side specification
- 2. NVECTOR module initialization
- 3. SUNLINSOL module initialization

Initialize one of the iterative SUNLINSOL modules, by calling one of FSUNPCGINIT, FSUNSPBCGSINIT, FSUNSPFGMRINIT, FSUNSPFGMRINIT or FSUNSPTFQMRINIT.

- 4. Sunnonlinsol module initialization
- 5. Problem specification
- 6. Set optional inputs

7. Linear solver interface specification

First, initialize the CVLS linear solver interface by calling FCVLSINIT.

Then, to initialize the CVBANDPRE preconditioner, make the following call:

```
CALL FCVBPINIT(NEQ, MU, ML, IER)
```

The arguments are as follows. NEQ is the problem size. MU and ML are the upper and lower half-bandwidths of the band matrix that is retained as an approximation of the Jacobian. IER is a return completion flag. A value of 0 indicates success, while a value of -1 indicates that a memory failure occurred.

Optionally, to specify that CVLs should use the supplied FCVJTIMES and FCVJTSETUP, make the call

```
CALL FCVLSSETJAC(FLAG, IER)
```

with FLAG $\neq 0$ (see step 8 in §5.2.4 for details).

- 8. Nonlinear solver interface specification
- 9. Problem solution
- 10. Additional solution output

11. CVBANDPRE **Optional outputs**

Optional outputs specific to the CVLS solver interface are listed in Table 5.4. To obtain the optional outputs associated with the CVBANDPRE module, make the following call:

CALL FCVBPOPT (LENRWBP, LENIWBP, NFEBP)

The arguments should be consistent with C type long int. Their returned values are as follows: LENRWBP is the length of real preconditioner work space, in realtype words. LENIWBP is the length of integer preconditioner work space, in integer words. NFEBP is the number of f(t, y) evaluations (calls to FCVFUN) for difference-quotient banded Jacobian approximations.

12. Memory deallocation

(The memory allocated for the FCVBP module is deallocated automatically by FCVFREE.)

5.2.8 Usage of the FCVBBD interface to CVBBDPRE

The FCVBBD interface sub-module is a package of C functions which, as part of the FCVODE interface module, support the use of the CVODE solver with the parallel NVECTOR_PARALLEL module, and the combination of the CVBBDPRE preconditioner module (see §4.7.2) with any of the Krylov iterative linear solvers.

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

- FCVBBDINIT interfaces to CVBBDPrecInit.
- FCVBBDREINIT interfaces to CVBBDPrecReInit.
- FCVBBDOPT interfaces to CVBBDPRE optional output functions.

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

FCVBBD routine (FORTRAN, user-supplied)	CVODE function (C, interface)	CVODE type of interface function
FCVLOCFN	FCVgloc	CVLocalFn
FCVCOMMF	FCVcfn	CVCommFn
FCVJTIMES	FCVJtimes	CVLsJacTimesVecFn
FCVJTSETUP	FCVJTSetup	CVLsJacTimesSetupFn

As with the rest of the FCVODE 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 §5.2.3, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file fcvbbd.h.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in §5.2.4 are graved-out.

- 1. Right-hand side specification
- 2. NVECTOR module initialization

3. SUNLINSOL module initialization

Initialize one of the iterative SUNLINSOL modules, by calling one of FSUNPCGINIT, FSUNSPBCGSINIT, FSUNSPFGMRINIT, FSUNSPFGMRINIT.

- 4. SUNNONLINSOL module initialization
- 5. Problem specification
- 6. Set optional inputs

7. Linear solver interface specification

First, initialize the CVLS iterative linear solver interface by calling FCVLSINIT.

Then, to initialize the CVBBDPRE preconditioner, make the following call:

```
CALL FCVBBDINIT(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 g for difference quotients (optional). A value of 0.0 indicates the default, g unit roundoff. IER is a return completion flag. A value of 0 indicates success, while a value of g indicates that a memory failure occurred or that an input had an illegal value.

Optionally, to specify that SPGMR, SPBCGS, or SPTFQMR should use the supplied FCVJTIMES, make the call

```
CALL FCVLSSETJAC(FLAG, IER)
```

with FLAG $\neq 0$ (see step 8 in §5.2.4 for details).

- 8. Nonlinear solver interface specification
- 9. Problem solution
- 10. Additional solution output

11. CVBBDPRE **Optional outputs**

Optional outputs specific to the CVLS solver interface are listed in Table 5.4. To obtain the optional outputs associated with the CVBBDPRE module, make the following call:

```
CALL FCVBBDOPT (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. These sizes are local to the current processor. NGEBBD is the number of g(t, y) evaluations (calls to FCVLOCFN) 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 CVBBDPRE preconditioner, then the CVODE package can be re-initialized for the second and subsequent problems by calling FCVREINIT, following which a call to FCVBBDINIT may or may not be needed. If the input arguments are the same, no FCVBBDINIT 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 FCVBBDREINIT(NLOCAL, MUDQ, MLDQ, DQRELY, IER)
```

This reinitializes the CVBBDPRE preconditioner, but without reallocating its memory. The arguments of the FCVBBDREINIT routine have the same names and meanings as those of FCVBBDINIT. If the value of MU or ML is being changed, then a call to FCVBBDINIT must be made. Finally, if there is a change in any of the linear solver inputs, then a call to one of FSUNPCGINIT, FSUNSPECGSINIT, FSUNSPFGMRINIT or FSUNSPTFQMRINIT, followed by a call to FCVLSINIT must also be made; in this case the linear solver memory is reallocated.

13. Memory deallocation

(The memory allocated for the FCVBBD module is deallocated automatically by FCVFREE.)

14. User-supplied routines

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

```
SUBROUTINE FCVGLOCFN (NLOC, T, YLOC, GLOC, IPAR, RPAR, IER)
DIMENSION YLOC(*), GLOC(*), IPAR(*), RPAR(*)
```

This routine is to evaluate the function g(t,y) approximating f (possibly identical to f), in terms of T=t, and the array YLOC (of length NLOC), which is the sub-vector of y local to this processor. The resulting (local) sub-vector is to be stored in the array GLOC. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if FCVGLOCFN failed unrecoverably (in which case the integration is halted).

```
SUBROUTINE FCVCOMMFN (NLOC, T, YLOC, IPAR, RPAR, IER) DIMENSION YLOC(*), IPAR(*), RPAR(*)
```

This routine is to perform the inter-processor communication necessary for the FCVGLOCFN routine. Each call to FCVCOMMFN is preceded by a call to the right-hand side routine FCVFUN with the same arguments T and YLOC. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. IER is an error return flag (currently not used; set IER=0). Thus FCVCOMMFN can omit any communications done by FCVFUN if relevant to the evaluation of GLOC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if FCVCOMMFN failed unrecoverably (in which case the integration is halted).



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

Optionally, the user can supply routines FCVJTIMES and FCVJTSETUP for the evaluation of Jacobian-vector products, as described above in step 8 in §5.2.4.

Chapter 6

CVODE Features for GPU Accelerated Computing

This chapter is concerned with using GPU-acceleration and CVODE for the solution of IVPs.

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)	√			√	√
NVECTOR_RAJA (§7.11)	√			√	🗸
NVECTOR_OPENMPDEV (§7.13)	√	✓	√	√	
SUNMATRIX_CUSPARSE (§8.7)	✓			✓	🗸
SUNLINSOL_CUSOLVERSP_BATCHQR ($\S 9.12$)	√			✓	 √
SUNLINSOL_SPGMR ($\S9.14$)	†	†	†	†	†
SUNLINSOL_SPFGMR ($\S9.15$)	†	†	†	†	†
SUNLINSOL_SPTFQMR ($\S9.17$)	†	†	†	†	†
SUNLINSOL_SPBCGS (§9.16)	†	†	†	†	†
SUNLINSOL_PCG ($\S9.18$)	†	†	†	†	†
SUNNONLINSOL_NEWTON (§10.3)	†	†	†	†	†
SUNNONLINSOL_FIXEDPOINT (§10.4)	†	†	†	†	†

6.2 Steps for Using GPU Accelerated SUNDIALS

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

- 1. Utilize a GPU-enabled 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.

 $F2003\ Name\ FN_VClone$

$N_{\text{VCloneEmpty}}$

Call v = N_VCloneEmpty(w);

Description Creates a new N_Vector of the same type as an existing vector w and sets the ops field.

It does not allocate storage for data.

Arguments w (N_Vector) a NVECTOR object

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

return NULL.

F2003 Name FN_VCloneEmpty

N_VDestroy

Call N_VDestroy(v);

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

Arguments v (N_Vector) a NVECTOR object to destroy

Return value None

F2003 Name FN_VDestroy

N_VSpace

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

Description Returns storage requirements for one N_Vector. 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 N_Vector is a contiguous array of realtype and is

accessible from the device (e.g., GPU).

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

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 ${\tt x}$

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 ℓ_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 ℓ_1 norm of the N_Vector x: $m = \sum_{i=0}^{n-1} |x_i|$. Arguments x (N_Vector) a NVECTOR object to obtain the norm of

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

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

$|N_{V}$ ConstVectorArray

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 \mathbf{x} and \mathbf{y} :

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

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

Arguments x (N_Vector) a NVECTOR object

y (N_Vector) a NVECTOR object

Return value realtype

 $F2003\ Name\ FN_VDotProdLocal$

N_VMaxNormLocal

 $Description \quad This \ routine \ computes \ the \ MPI \ task-local \ portion \ of \ the \ maximum \ norm \ of \ the \ \texttt{N_Vector}$

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:

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

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

Arguments x (N_Vector) a NVECTOR object

Return value realtype

F2003 Name FN_VMinLocal

N_VL1NormLocal

Call n = N_VL1NormLocal(x);

Description This routine computes the MPI task-local portion of the ℓ_1 norm of the N_Vector x:

$$n = \sum_{i=0}^{n_{local}-1} |x_i|,$$

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

Arguments x (N_Vector) a NVECTOR object

Return value realtype

F2003 Name FN_VL1NormLocal

N_VWSqrSumLocal

Call s = N_VWSqrSumLocal(x,w);

Description This routine computes the MPI task-local portion of the weighted squared sum of the $N_{\text{-}}$ 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 numi

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 **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 \mathtt{buf} into the vector \mathtt{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);
```

```
realtype
               (*nvmaxnormlocal)(N_Vector);
  realtype
               (*nvminlocal)(N_Vector);
  realtype
               (*nvl1normlocal)(N_Vector);
  booleantype
               (*nvinvtestlocal)(N_Vector, N_Vector);
  booleantype
               (*nvconstrmasklocal)(N_Vector, N_Vector, N_Vector);
               (*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 CVODE

In Table 7.2 below, we list the vector functions in the NVECTOR module used within the CVODE package. The table also shows, for each function, which of the code modules uses the function. The CVODE column shows function usage within the main integrator module, while the remaining

columns show function usage within each of the CVODE linear solver interfaces, the CVBANDPRE and CVBBDPRE preconditioner modules, and the FCVODE module. Here CVLS stands for the generic linear solver interface in CVODE, and CVDIAG stands for the diagonal linear solver interface in CVODE.

At this point, we should emphasize that the CVODE user does not need to know anything about the usage of vector functions by the CVODE code modules in order to use CVODE. The information is presented as an implementation detail for the interested reader.

CVBANDPRI CVBBDPRE FCVODE CVDIAG CVODE N_VGetVectorID N_VGetLength 4 N_{VClone} N_VCloneEmpty 1 \checkmark N_VDestroy \checkmark 2 N_VSpace N_VGetArrayPointer 1 N_VSetArrayPointer 1 **√** N_VLinearSum N_VConst N_VProd \checkmark N_{VDiv} N_VScale N_{VAbs} ✓ N_{VInv} $N_{VAddConst}$ $N_{VMaxNorm}$ \checkmark N_VWrmsNorm $N_{-}VMin$ $N_VMinQuotient$ $\sqrt{}$ $N_VConstrMask$ $N_VCompare$ \checkmark $N_{VInvTest}$ ✓ $N_{-}VLinearCombination$ N_VScaleAddMulti N_VDotProdMulti 3 3 N_VScaleVectorArray

Table 7.2: List of vector functions usage by CVODE 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 CVODE modules for user feedback.
- 3. The optional function N_VDotProdMulti is only used in the SUNNONLINSOL_FIXEDPOINT module, or when Classical Gram-Schmidt is enabled with SPGMR or SPFGMR. The remaining operations from Sections 7.1.2 and 7.1.3 not listed above are unused and a user-supplied NVECTOR module for CVODE could omit these operations.

4. This routine is only used when an iterative or matrix iterative SUNLINSOL module is supplied to CVODE.

Each SUNLINSOL object may require additional NVECTOR routines not listed in the table above. Please see the the relevant descriptions of these modules in Sections 9.5-9.18 for additional detail on their NVECTOR requirements.

The vector functions listed in Section 7.1.1 that are *not* used by CVODE are: N_VWL2Norm, N_VDotProd, N_VL1Norm, N_VWrmsNormMask, and N_VGetCommunicator. Therefore, a user-supplied NVECTOR module for CVODE could omit these functions (although some may be needed by SUNNONLINSOL or SUNLINSOL modules). The functions N_MinQuotient, N_VConstrMask, and N_VCompare are only used when constraint checking is enabled and may be omitted if this feature is not used.

7.3 The NVECTOR_SERIAL implementation

The serial implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_SERIAL, defines the *content* field of N_Vector to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, and a boolean flag *own_data* which specifies the ownership of *data*.

```
struct _N_VectorContent_Serial {
   sunindextype length;
   booleantype own_data;
   realtype *data;
};
```

The header file to include when using this module is nvector_serial.h. The installed module library to link to is libsundials_nvecserial.lib where .lib is typically .so for shared libraries and .a for static libraries.

7.3.1 NVECTOR_SERIAL accessor macros

The following macros are provided to access the content of an NVECTOR_SERIAL vector. The suffix _S in the names denotes the serial version.

NV_CONTENT_S

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

The assignment $v_cont = NV_CONTENT_S(v)$ sets v_cont to be a pointer to the serial N_Vector content structure.

Implementation:

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

• NV_OWN_DATA_S, NV_DATA_S, NV_LENGTH_S

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

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

The assignment $v_len = NV_LENGTH_S(v)$ sets v_len to be the length of v. On the other hand, the call $NV_LENGTH_S(v) = len_v$ sets the length of v to be len_v .

Implementation:

```
#define NV_OWN_DATA_S(v) ( NV_CONTENT_S(v)->own_data )
#define NV_DATA_S(v) ( NV_CONTENT_S(v)->data )
#define NV_LENGTH_S(v) ( NV_CONTENT_S(v)->length )
```

• NV_Ith_S

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

The assignment $r = NV_{i,i}$ 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 module.

N_VNewEmpty_Serial

Prototype N_Vector N_VNewEmpty_Serial(sunindextype vec_length);

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

F2003 Name This function is callable as FN_VNewEmpty_Serial when using the Fortran 2003 interface module.

N_VMake_Serial

Prototype N_Vector N_VMake_Serial(sunindextype vec_length, realtype *v_data);

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

(This function does *not* allocate memory for v_data itself.)

F2003 Name This function is callable as FN_VMake_Serial when using the Fortran 2003 interface module.

N_VCloneVectorArray_Serial

Prototype N_Vector *N_VCloneVectorArray_Serial(int count, N_Vector w);

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

F2003 Name This function is callable as FN_VCloneVectorArray_Serial when using the Fortran 2003 interface module.

N_VCloneVectorArrayEmpty_Serial

Prototype N_Vector *N_VCloneVectorArrayEmpty_Serial(int count, N_Vector w);

Description This function creates (by cloning) an array of count serial vectors, each with an empty

(NULL) data array.

F2003 Name This function is callable as FN_VCloneVectorArrayEmpty_Serial when using the For-

tran 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 Fortran 2003 interface module.

N_VEnableScaleAddMulti_Serial

Prototype int N_VEnableScaleAddMulti_Serial(N_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableScaleAddMulti_Serial when using the Fortran 2003 interface module.

N_VEnableDotProdMulti_Serial

Prototype int N_VEnableDotProdMulti_Serial(N_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableDotProdMulti_Serial when using the Fortran 2003 interface module.

N_VEnableLinearSumVectorArray_Serial

Prototype int N_VEnableLinearSumVectorArray_Serial(N_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableLinearSumVectorArray_Serial when using the Fortran 2003 interface module.

N_VEnableScaleVectorArray_Serial

Prototype int N_VEnableScaleVectorArray_Serial(N_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableScaleVectorArray_Serial when using the Fortran 2003 interface module.

N_VEnableConstVectorArray_Serial

Prototype int N_VEnableConstVectorArray_Serial(N_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableConstVectorArray_Serial when using the Fortran 2003 interface module.

N_VEnableWrmsNormVectorArray_Serial

Prototype int N_VEnableWrmsNormVectorArray_Serial(N_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableWrmsNormVectorArray_Serial when using the Fortran 2003 interface module.

N_VEnableWrmsNormMaskVectorArray_Serial

Prototype int N_VEnableWrmsNormMaskVectorArray_Serial(N_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableWrmsNormMaskVectorArray_Serial when using the Fortran 2003 interface module.

N_VEnableScaleAddMultiVectorArray_Serial

Prototype int N_VEnableScaleAddMultiVectorArray_Serial(N_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearCombinationVectorArray_Serial

Prototype int N_VEnableLinearCombinationVectorArray_Serial(N_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the serial vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

Notes

- When looping over the components of an N_Vector v, it is more efficient to first obtain the component array via v_data = NV_DATA_S(v) and then access v_data[i] within the loop than it is to use NV_Ith_S(v,i) within the loop.
- N_VNewEmpty_Serial, N_VMake_Serial, and N_VCloneVectorArrayEmpty_Serial set the field $own_data = SUNFALSE$. N_VDestroy_Serial and N_VDestroyVectorArray_Serial will not attempt to free the pointer data for any N_Vector with own_data set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the data pointer.
- To maximize efficiency, vector operations in the NVECTOR_SERIAL implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.





7.3.3 NVECTOR SERIAL Fortran interfaces

The NVECTOR_SERIAL module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The fnvector_serial_mod FORTRAN module defines interfaces to all NVECTOR_SERIAL C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function N_VNew_Serial is interfaced as FN_VNew_Serial.

The FORTRAN 2003 NVECTOR_SERIAL interface module can be accessed with the use statement, i.e. use fnvector_serial_mod, and linking to the library libsundials_fnvectorserial_mod.lib in addition to the C library. For details on where the library and module file fnvector_serial_mod.mod are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the libsundials_fnvectorserial_mod library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the NVECTOR_SERIAL module also includes a FORTRAN-callable function 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 Tth P

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

The assignment $r = NV_i(v,i)$ sets r to be the value of the i-th component of the local part of v. The assignment $NV_i(v,i) = r$ sets the value of the i-th component of the local part of v to be r.

Here i ranges from 0 to n-1, where n is the local length.

Implementation:

```
#define NV_Ith_P(v,i) ( NV_DATA_P(v)[i] )
```

7.4.2 NVECTOR_PARALLEL functions

The NVECTOR_PARALLEL module defines parallel implementations of all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4. Their names are obtained from those in these tables by appending the suffix _Parallel (e.g. N_VDestroy_Parallel). The module NVECTOR_PARALLEL provides the following additional user-callable routines:

N_VNew_Parallel

```
Prototype N_Vector N_VNew_Parallel(MPI_Comm comm, sunindextype local_length, sunindextype global_length);
```

Description This function creates and allocates memory for a parallel vector.

F2003 Name This function is callable as FN_VNew_Parallel when using the Fortran 2003 interface module.

N_VNewEmpty_Parallel

Prototype N_Vector N_VNewEmpty_Parallel(MPI_Comm comm, sunindextype local_length, sunindextype global_length);

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

F2003 Name This function is callable as FN_VNewEmpty_Parallel when using the Fortran 2003 interface module.

N_VMake_Parallel

Prototype N_Vector N_VMake_Parallel(MPI_Comm comm, sunindextype local_length, sunindextype global_length, 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 int N_VEnableScaleAddMultiVectorArray_Parallel(N_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearCombinationVectorArray_Parallel

Prototype int N_VEnableLinearCombinationVectorArray_Parallel(N_Vector v, booleantype tf);

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the parallel vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

Notes

- When looping over the components of an N_Vector v, it is more efficient to first obtain the local component array via v_data = NV_DATA_P(v) and then access v_data[i] within the loop than it is to use NV_Ith_P(v,i) within the loop.
- N_VNewEmpty_Parallel, N_VMake_Parallel, and N_VCloneVectorArrayEmpty_Parallel set the field own_data = SUNFALSE. N_VDestroy_Parallel and N_VDestroyVectorArray_Parallel will not attempt to free the pointer data for any N_Vector with own_data set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the data pointer.
- To maximize efficiency, vector operations in the NVECTOR_PARALLEL implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

7.4.3 NVECTOR_PARALLEL Fortran interfaces

For solvers that include a FORTRAN 77 interface module, the NVECTOR_PARALLEL module also includes a FORTRAN-callable function 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 $r = NV_{int}(v,i)$ sets r to be the value of the i-th component of v. The assignment $NV_{int}(v,i) = r$ 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)

created with N_VCloneVectorArray_OpenMP or with N_VCloneVectorArrayEmpty_OpenMP.

F2003 Name This function is callable as FN_VDestroyVectorArray_OpenMP when using the Fortran

2003 interface module.

N_VPrint_OpenMP

Prototype void N_VPrint_OpenMP(N_Vector v)

Description This function prints the content of an OpenMP vector to stdout.

F2003 Name This function is callable as FN_VPrint_OpenMP when using the Fortran 2003 interface

module.

N_VPrintFile_OpenMP

Prototype void N_VPrintFile_OpenMP(N_Vector v, FILE *outfile)

Description This function prints the content of an OpenMP vector to outfile.

 $F2003 \ \ Name \ \ This function is callable as {\tt FN_VPrintFile_OpenMP} \ when using the \ Fortran \ 2003 \ interface$

module.

By default all fused and vector array operations are disabled in the NVECTOR_OPENMP module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with N_VNew_OpenMP, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using N_VClone. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with N_VNew_OpenMP will have the default settings for the NVECTOR_OPENMP module.

N_VEnableFusedOps_OpenMP

Prototype int N_VEnableFusedOps_OpenMP(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array op-

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 0 for success and -1 if the input

vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableLinearCombination_OpenMP when using the 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

Prototype int N_VEnableWrmsNormVectorArray_OpenMP(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableWrmsNormVectorArray_OpenMP when using the Fortran 2003 interface module.

| N_VEnableWrmsNormMaskVectorArray_OpenMP

Prototype int N_VEnableWrmsNormMaskVectorArray_OpenMP(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

F2003 Name This function is callable as FN_VEnableWrmsNormMaskVectorArray_OpenMP when using the Fortran 2003 interface module.

N_VEnableScaleAddMultiVectorArray_OpenMP

Prototype int N_VEnableScaleAddMultiVectorArray_OpenMP(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearCombinationVectorArray_OpenMP

Prototype int N_VEnableLinearCombinationVectorArray_OpenMP(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the OpenMP vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

Notes

- When looping over the components of an N_Vector v, it is more efficient to first obtain the component array via v_data = NV_DATA_OMP(v) and then access v_data[i] within the loop than it is to use NV_Ith_OMP(v,i) within the loop.
- N_VNewEmpty_OpenMP, N_VMake_OpenMP, and N_VCloneVectorArrayEmpty_OpenMP set the field $own_data = SUNFALSE$. N_VDestroy_OpenMP and N_VDestroyVectorArray_OpenMP will not attempt to free the pointer data for any N_Vector with own_data set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the data pointer.
- To maximize efficiency, vector operations in the NVECTOR_OPENMP implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

7.5.3 NVECTOR_OPENMP Fortran interfaces

The NVECTOR_OPENMP module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The nvector_openmp_mod FORTRAN module defines interfaces to most NVECTOR_OPENMP C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function N_VNew_OpenMP is interfaced as FN_VNew_OpenMP.





The FORTRAN 2003 NVECTOR_OPENMP interface module can be accessed with the use statement, i.e. use fnvector_openmp_mod, and linking to the library libsundials_fnvectoropenmp_mod.lib in addition to the C library. For details on where the library and module file fnvector_openmp_mod.mod are installed see Appendix A.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the NVECTOR_OPENMP module also includes a FORTRAN-callable function 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

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.

${\tt N_VEnableLinearSumVectorArray_Pthreads}$

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.

${\tt N_VEnableWrmsNormMaskVectorArray_Pthreads}$

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 [38] and the ark_diurnal_kry_ph.c example program for ARKODE [48].

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_{V} NewEmpty_ParHyp

Prototype N_Vector N_VNewEmpty_ParHyp(MPI_Comm comm, sunindextype local_length, sunindextype global_length)

Description This function creates a new parhyp N_Vector with the pointer to the hypre vector set to NULL.

N_VMake_ParHyp

Prototype N_Vector N_VMake_ParHyp(HYPRE_ParVector x)

Description This function creates an N_Vector wrapper around an existing hypre parallel vector. It does not allocate memory for x itself.

N_VGetVector_ParHyp

Prototype HYPRE_ParVector N_VGetVector_ParHyp(N_Vector v)

Description This function returns the underlying hypre vector.

N_VCloneVectorArray_ParHyp

Prototype N_Vector *N_VCloneVectorArray_ParHyp(int count, N_Vector w)

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

N_VCloneVectorArrayEmpty_ParHyp

Prototype N_Vector *N_VCloneVectorArrayEmpty_ParHyp(int count, N_Vector w)

Description This function creates (by cloning) an array of count parallel vectors, each with an empty

(NULL) data array.

N_VDestroyVectorArray_ParHyp

Prototype void N_VDestroyVectorArray_ParHyp(N_Vector *vs, int count)

Description This function frees memory allocated for the array of count variables of type N_Vector created with N_VCloneVectorArray_ParHyp or with N_VCloneVectorArrayEmpty_ParHyp.

N_VPrint_ParHyp

Prototype void N_VPrint_ParHyp(N_Vector v)

Description This function prints the local content of a parhyp vector to stdout.

$N_VPrintFile_ParHyp$

Prototype void N_VPrintFile_ParHyp(N_Vector v, FILE *outfile)

Description This function prints the local content of a parhyp vector to outfile.

By default all fused and vector array operations are disabled in the NVECTOR_PARHYP module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with N_VMake_ParHyp, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using N_VClone. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with N_VMake_ParHyp will have the default settings for the NVECTOR_PARHYP module.

N_VEnableFusedOps_ParHyp

Prototype int N_VEnableFusedOps_ParHyp(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearCombination_ParHyp

Prototype int N_VEnableLinearCombination_ParHyp(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableScaleAddMulti_ParHyp

Prototype int N_VEnableScaleAddMulti_ParHyp(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableDotProdMulti_ParHyp

Prototype int N_VEnableDotProdMulti_ParHyp(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearSumVectorArray_ParHyp

Prototype int N_VEnableLinearSumVectorArray_ParHyp(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableScaleVectorArray_ParHyp

Prototype int N_VEnableScaleVectorArray_ParHyp(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector

arrays in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableConstVectorArray_ParHyp

Prototype int N_VEnableConstVectorArray_ParHyp(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector

arrays in the parhyp vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableWrmsNormVectorArray_ParHyp

Prototype int N_VEnableWrmsNormVectorArray_ParHyp(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for

vector arrays in the parhyp vector. The return value is ${\tt 0}$ 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)

booledmoype of

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

Description

- 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 [37].

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 $\ensuremath{\mathsf{ops}}$ 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.

${\tt N_VEnableLinearCombinationVectorArray_Petsc}$

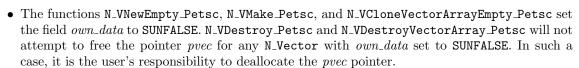
Prototype int N_VEnableLinearCombinationVectorArray_Petsc(N_Vector v, booleantype tf)

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 [38].

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)

 $Description \quad This \ function \ creates \ and \ allocates \ memory \ for \ a \ {\tt CUDA} \ {\tt N_Vector}. \ The \ vector \ data \ array$

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

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 N_VNewWithMemHelp_Cuda instead.

The module NVECTOR_CUDA also provides the following user-callable routines:



N_VSetKernelExecPolicy_Cuda

Prototype void N_VSetKernelExecPolicy_Cuda(N_Vector v, SUNCudaExecPolicy* stream_exec_policy, SUNCudaExecPolicy* reduce_exec_policy);

Description

This function sets the execution policies which control the kernel parameters utilized when launching the streaming and reduction CUDA kernels. By default the vector is setup to use the SUNCudaThreadDirectExecPolicy and SUNCudaBlockReduceExecPolicy. Any custom execution policy for reductions must ensure that the grid dimensions (number of thread blocks) is a multiple of the CUDA warp size (32). See section 7.9.2 below for more information about the SUNCudaExecPolicy class.

Note: All vectors used in a single instance of a SUNDIALS solver must use the same execution policy. It is **strongly recommended** that this function is called immediately after constructing the vector, and any subsequent vector be created by cloning to ensure consistent execution policies across vectors.

N_VSetCudaStream_Cuda

Prototype void N_VSetCudaStream_Cuda(N_Vector v, cudaStream_t *stream)

Description This function sets the CUDA stream that all vector kernels will be launched on. By default an NVECTOR_CUDA uses the default CUDA stream.

Note: All vectors used in a single instance of a SUNDIALS solver must use the same CUDA stream. It is **strongly recommended** that this function is called immediately after constructing the vector, and any subsequent vector be created by cloning to ensure consistent execution policies across vectors.

This function will be removed in the next major release, user should utilize the N_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)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableLinearSumVectorArray_Cuda

Prototype int N_VEnableLinearSumVectorArray_Cuda(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableScaleVectorArray_Cuda

Prototype int N_VEnableScaleVectorArray_Cuda(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableConstVectorArray_Cuda

Prototype int N_VEnableConstVectorArray_Cuda(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the CUDA vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

N_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)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused

operation in the HIP vector. The return value is 0 for success and -1 if the input vector

or its ops structure are NULL.

N_VEnableLinearSumVectorArray_Hip

Prototype int N_VEnableLinearSumVectorArray_Hip(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for

vector arrays in the HIP vector. The return value is 0 for success and -1 if the input

vector or its ops structure are NULL.

N_VEnableScaleVectorArray_Hip

Prototype int N_VEnableScaleVectorArray_Hip(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector

arrays in the HIP vector. The return value is $\hat{0}$ 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 [38].

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.

```
{\tt 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 operations in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1

if the input vector or its ops structure are NULL.

N_VEnableLinearCombination_OpenMPDEV

Prototype int N_VEnableLinearCombination_OpenMPDEV(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused

operation in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and

-1 if the input vector or its ops structure are NULL.

N_VEnableScaleAddMulti_OpenMPDEV

Prototype int N_VEnableScaleAddMulti_OpenMPDEV(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to

multiple vectors fused operation in the NVECTOR_OPENMPDEV vector. The return value

is 0 for success and -1 if the input vector or its ops structure are NULL.

N_VEnableDotProdMulti_OpenMPDEV

Prototype int N_VEnableDotProdMulti_OpenMPDEV(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and

-1 if the input vector or its ops structure are NULL.

N_VEnableLinearSumVectorArray_OpenMPDEV

Prototype int N_VEnableLinearSumVectorArray_OpenMPDEV(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the NVECTOR_OPENMPDEV vector. The return value is 0 for success

and -1 if the input vector or its ops structure are NULL.

| N_VEnableScaleVectorArray_OpenMPDEV

Prototype int N_VEnableScaleVectorArray_OpenMPDEV(N_Vector v, booleantype tf)

 $\hbox{Description This function enables ($\tt SUNTRUE)$ or disables ($\tt SUNFALSE)$ the scale operation for vector } \\$

arrays in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if the input vector or its ops structure are NULL.

| N_VEnableConstVectorArray_OpenMPDEV

Prototype int N_VEnableConstVectorArray_OpenMPDEV(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector

arrays in the NVECTOR_OPENMPDEV vector. The return value is 0 for success and -1 if

the input vector or its ops structure are NULL.

N_VEnableWrmsNormVectorArray_OpenMPDEV

Prototype int N_VEnableWrmsNormVectorArray_OpenMPDEV(N_Vector v, booleantype tf)

Description This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for

vector arrays in the NVECTOR_OPENMPDEV vector. The return value is 0 for success

and -1 if the input vector or its ops structure are NULL.

N_VEnableWrmsNormMaskVectorArray_OpenMPDEV

 $\label{lem:prototype} Prototype \quad \text{ int N_VEnableWrmsNormMaskVectorArray_OpenMPDEV(N_Vector v,}$

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 [37].

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.

$N_VEnableLinearCombination_ManyVector$

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.

Note: If SUNDIALS is configured with MPI disabled, then the mpiplusx library will not be built. Furthermore, any user codes that include nvector_mpiplusx.h *must* be compiled using an MPI-aware compiler.



7.17.2 NVECTOR MPIPLUSX functions

The NVECTOR_MPIPLUSX module adopts all vector operations listed in Tables 7.1.1, 7.1.2, 7.1.3, and 7.1.4, from the NVECTOR_MPIMANYVECTOR (see section 7.16.2) except for N_VGetArrayPointer and N_VSetArrayPointer; the module provides its own implementation of these functions that call the local vector implementations. Therefore, the NVECTOR_MPIPLUSX module implements all of the operations listed in the referenced sections except for N_VScaleAddMultiVectorArray, and N_VLinearCombinationVectorArray Accordingly, 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);

Description This function sets the data array pointer for the local vector if the local vector implements the N_VSetArrayPointer operation.

F2003 Name This function is callable as FN_VSetArrayPointer_MPIPlusX when using the Fortran 2003 interface module.

The NVECTOR_MPIPLUSX module does not implement any fused or vector array operations. Instead users should enable/disable fused operations on the local vector.

Notes



- N_VMake_MPIPlusX sets the field $own_{-}data = SUNFALSE$. and N_VDestroy_MPIPlusX will not call N_VDestroy on the local vector. In this case, it is the user's responsibility to deallocate the local vector.
- To maximize efficiency, arithmetic vector operations in the NVECTOR_MPIPLUSX implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same local vector representations.

7.18 **NVECTOR** Examples

There are NVector examples that may be installed for the implementations provided with SUNDIALS. Each implementation makes use of the functions in test_nvector.c. These example functions show simple usage of the NVector family of functions. The input to the examples are the vector length, number of threads (if threaded implementation), and a print timing flag. The following is a list of the example functions in test_nvector.c:

• Test_N_VClone: Creates clone of vector and checks validity of clone.



- Test_N_VCloneEmpty: Creates clone of empty vector and checks validity of clone.
- Test_N_VCloneVectorArray: Creates clone of vector array and checks validity of cloned array.
- Test_N_VCloneVectorArray: Creates clone of empty vector array and checks validity of cloned array.
- Test_N_VGetArrayPointer: Get array pointer.
- Test_N_VSetArrayPointer: Allocate new vector, set pointer to new vector array, and check values.
- Test_N_VGetLength: Compares self-reported length to calculated length.
- Test_N_VGetCommunicator: Compares self-reported communicator to the one used in constructor; or for MPI-unaware vectors it ensures that NULL is reported.
- Test_N_VLinearSum Case 1a: Test y = x + y
- Test_N_VLinearSum Case 1b: Test y = -x + y
- Test_N_VLinearSum Case 1c: Test y = ax + y
- Test_N_VLinearSum Case 2a: Test x = x + y
- Test_N_VLinearSum Case 2b: Test x = x y
- Test_N_VLinearSum Case 2c: Test x = x + by
- Test_N_VLinearSum Case 3: Test z = x + y
- Test_N_VLinearSum Case 4a: Test z = x y
- Test_N_VLinearSum Case 4b: Test z = -x + y
- Test_N_VLinearSum Case 5a: Test z = x + by
- Test_N_VLinearSum Case 5b: Test z = ax + y
- Test_N_VLinearSum Case 6a: Test z = -x + by
- Test_N_VLinearSum Case 6b: Test z = ax y
- Test_N_VLinearSum Case 7: Test z = a(x + y)
- Test_N_VLinearSum Case 8: Test z = a(x y)
- Test_N_VLinearSum Case 9: Test z = ax + by
- Test_N_VConst: Fill vector with constant and check result.
- Test_N_VProd: Test vector multiply: z = x * y
- Test_N_VDiv: Test vector division: z = x / y
- Test_N_VScale: Case 1: scale: x = cx
- Test_N_VScale: Case 2: copy: z = x
- Test_N_VScale: Case 3: negate: z = -x
- Test_N_VScale: Case 4: combination: z = cx
- Test_N_VAbs: Create absolute value of vector.

- Test_N_VAddConst: add constant vector: z = c + x
- Test_N_VDotProd: Calculate dot product of two vectors.
- Test_N_VMaxNorm: Create vector with known values, find and validate the max norm.
- Test_N_VWrmsNorm: Create vector of known values, find and validate the weighted root mean square.
- Test_N_VWrmsNormMask: Create vector of known values, find and validate the weighted root mean square using all elements except one.
- Test_N_VMin: Create vector, find and validate the min.
- Test_N_VWL2Norm: Create vector, find and validate the weighted Euclidean L2 norm.
- Test_N_VL1Norm: Create vector, find and validate the L1 norm.
- Test_N_VCompare: Compare vector with constant returning and validating comparison vector.
- Test_N_VInvTest: Test z[i] = 1 / x[i]
- Test_N_VConstrMask: Test mask of vector x with vector c.
- Test_N_VMinQuotient: Fill two vectors with known values. Calculate and validate minimum quotient.
- Test_N_VLinearCombination Case 1a: Test x = a x
- 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 1b: z = a x + y
- Test_N_VScaleAddMulti Case 2a: Y[i] = c[i] x + Y[i], i = 1,2,3
- Test_N_VScaleAddMulti Case 2b: Z[i] = c[i] x + Y[i], i = 1,2,3
- Test_N_VDotProdMulti Case 1: Calculate the dot product of two vectors
- Test_N_VDotProdMulti Case 2: Calculate the dot product of one vector with three other vectors in a vector array.
- Test_N_VLinearSumVectorArray Case 1: z = a x + b y
- Test_N_VLinearSumVectorArray Case 2a: Z[i] = a X[i] + b Y[i]
- Test_N_VLinearSumVectorArray Case 2b: X[i] = a X[i] + b Y[i]
- Test_N_VLinearSumVectorArray Case 2c: Y[i] = a X[i] + b Y[i]
- Test_N_VScaleVectorArray Case 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 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 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]
- ullet 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

 $F2003 \; Name \; FSUNMatScaleAdd$

SUNMatScaleAddI

Call ier = SUNMatScaleAddI(c, A);

Description Performs the operation A = cA + I.

Arguments c (realtype) constant that scales A

A (SUNMatrix) a SUNMATRIX object

Return value A SUNMATRIX return code of type int denoting success/failure

 $F2003 \; Name \; FSUNMatScaleAddI$

SUNMatMatvecSetup

Call ier = SUNMatMatvecSetup(A);

Description Performs any setup necessary to perform a matrix-vector product. It is useful for

SUNMatrix implementations which need to prepare the matrix itself, or communication

structures before performing the matrix-vector product.

Arguments A (SUNMatrix) a SUNMATRIX object

Return value A SUNMATRIX return code of type int denoting success/failure

F2003 Name FSUNMatMatvecSetup

SUNMatMatvec

Call ier = SUNMatMatvec(A, x, y);

Description Performs the matrix-vector product operation, y = Ax. It should only be called with

vectors x and y that are compatible with the matrix A - both in storage type and

dimensions.

Arguments A (SUNMatrix) a SUNMATRIX object

x (N_Vector) a NVECTOR object

y (N_Vector) an output NVECTOR object

Return value A SUNMATRIX return code of type int denoting success/failure

F2003 Name FSUNMatMatvec

8.1.2 SUNMatrix utility functions

To aid in the creation of custom SUNMATRIX modules the generic SUNMATRIX module provides two utility functions SUNMatNewEmpty and SUNMatVCopyOps.

SUNMatNewEmpty

Call A = SUNMatNewEmpty();

Description The function SUNMatNewEmpty allocates a new generic SUNMATRIX object and initializes

its content pointer and the function pointers in the operations structure to NULL.

Arguments None

Return value This function returns a SUNMatrix object. If an error occurs when allocating the object,

then this routine will return NULL.

F2003 Name FSUNMatNewEmpty

SUNMatFreeEmpty

Call SUNMatFreeEmpty(A);

Description This routine frees the generic SUNMatrix object, under the assumption that any implementation-

specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the ops pointer is NULL, and, if it is not, it will

free it as well.

Arguments A (SUNMatrix) a SUNMatrix object

Return value None

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

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.

Matrix	Serial	Parallel	OpenMP	pThreads	hypre	PETSC	CUDA	RAJA	User
Interface		(MPI)			Vec.	Vec.			Suppl.
Dense	✓		✓	✓					✓
Band	✓		✓	✓					✓
Sparse	✓		✓	✓					✓
SLUNRloc	✓	✓	✓	✓	✓	✓			✓
User supplied	✓	✓	✓	✓	✓	✓	✓	✓	✓

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 CVODE

In Table 8.4, we list the matrix functions in the SUNMATRIX module used within the CVODE package. The table also shows, for each function, which of the code modules uses the function. The main CVODE integrator does not call any SUNMATRIX functions directly, so the table columns are specific to the CVLS interface and the CVBANDPRE and CVBBDPRE preconditioner modules. We further note that the CVLS interface only utilizes these routines when supplied with a *matrix-based* linear solver, i.e., the SUNMATRIX object passed to CVodeSetLinearSolver was not NULL.

At this point, we should emphasize that the CVODE user does not need to know anything about the usage of matrix functions by the CVODE code modules in order to use CVODE. The information is presented as an implementation detail for the interested reader.

	CVLS	CVBANDPRE	CVBBDPRE
SUNMatGetID	√		
SUNMatClone	√		
SUNMatDestroy	√	√	√
SUNMatZero	√	√	√
SUNMatCopy	√	√	√
SUNMatScaleAddI	√	√	√
SUNMatSpace	†	†	†

Table 8.4: List of matrix functions usage by CVODE 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 CVODE are: SUNMatScaleAdd and SUNMatMatvec. Therefore a user-supplied SUNMATRIX module for CVODE could omit these functions.

We note that the CVBANDPRE and CVBBDPRE preconditioner modules are hard-coded to use the SUNDIALS-supplied band SUNMATRIX type, so the most useful information above for user-supplied SUNMATRIX implementations is the column relating the CVLS requirements.

8.3 The SUNMatrix_Dense implementation

The dense implementation of the SUNMATRIX module provided with SUNDIALS, SUNMATRIX_DENSE, defines the *content* field of SUNMatrix to be the following structure:

```
struct _SUNMatrixContent_Dense {
  sunindextype M;
  sunindextype N;
  realtype *data;
  sunindextype ldata;
  realtype **cols;
};
These entries of the content field contain the following information:
      - 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 \le M and 0 \le i \le 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/sunmatrix_dense.h. The SUNMA-
TRIX_DENSE module is accessible from all SUNDIALS solvers without linking to the
```

8.3.1 SUNMatrix_Dense accessor macros

libsundials_sunmatrixdense module library.

The following macros are provided to access the content of a SUNMATRIX_DENSE matrix. The prefix SM_{-} in the names denotes that these macros are for SUNMatrix implementations, and the suffix $_{-}D$ denotes that these are specific to the dense version.

• 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:

• 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 × 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 Sundial solver. Here code is an integer input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); M and N are the corresponding dense matrix construction arguments (declared to match C type long int); and ier is an error return flag equal to 0 for success and -1 for failure. Both code and ier are declared to match C type int. Additionally, when using ARKODE with a non-identity mass matrix, the FORTRAN-callable function FSUNDenseMassMatInit(M, N, ier) initializes this SUNMATRIX_DENSE module for storing the mass matrix.

8.4 The SUNMatrix_Band implementation

The banded implementation of the SUNMATRIX module provided with SUNDIALS, SUNMATRIX_BAND, defines the *content* field of SUNMatrix to be the following structure:

```
struct _SUNMatrixContent_Band {
   sunindextype M;
   sunindextype N;
   sunindextype mu;
   sunindextype ml;
   sunindextype s_mu;
   sunindextype ldim;
   realtype *data;
   sunindextype ldata;
   realtype **cols;
};
```

A diagram of the underlying data representation in a banded matrix is shown in Figure 8.1. A more complete description of the parts of this *content* field is given below:

```
{\tt M} - number of rows 
 {\tt N} - number of columns ({\tt N}={\tt M})
```

 $\mbox{mu} \quad \mbox{-upper half-bandwidth, } 0 \leq \mbox{mu} < \mbox{N}$

ml - lower half-bandwidth, $0 \le ml < N$

s_mu - storage upper bandwidth, $mu \le s_mu < N$. 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)
```

- 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) )
```



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.

• 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 )
#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

Prototype SUNMatrix SUNBandMatrix(sunindextype N, sunindextype mu, sunindextype ml)

Description This constructor function creates and allocates memory for a banded SUNMatrix. Its arguments are the matrix size, N, and the upper and lower half-bandwidths of the matrix, mu and ml. The stored upper bandwidth is set to mu+ml to accommodate subsequent factorization in the SUNLINSOL_BAND and SUNLINSOL_LAPACKBAND modules.

F2003 Name This function is callable as FSUNBandMatrix when using the Fortran 2003 interface module.

SUNBandMatrixStorage

Prototype SUNMatrix SUNBandMatrixStorage(sunindextype N, sunindextype mu, sunindextype ml, sunindextype smu)

Description This constructor function creates and allocates memory for a banded SUNMatrix. Its arguments are the matrix size, N, the upper and lower half-bandwidths of the matrix, mu and ml, and the stored upper bandwidth, smu. When creating a band SUNMatrix, this value should be

- at least min(N-1,mu+ml) if the matrix will be used by the SUNLINSOL_BAND module;
- exactly equal to mu+ml if the matrix will be used by the SUNLINSOL_LAPACKBAND module:
- at least mu if used in some other manner.

Note: it is strongly recommended that users call the default constructor, SUNBandMatrix, in all standard use cases. This advanced constructor is used internally within SUNDIALS solvers, and is provided to users who require banded matrices for non-default purposes.

SUNBandMatrix_Print

Prototype void SUNBandMatrix_Print(SUNMatrix A, FILE* outfile)

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

SUNBandMatrix_Rows

Prototype sunindextype SUNBandMatrix_Rows(SUNMatrix A)

Description This function returns the number of rows in the banded SUNMatrix.

F2003 Name This function is callable as FSUNBandMatrix_Rows when using the Fortran 2003 interface module.

SUNBandMatrix_Columns

Prototype sunindextype SUNBandMatrix_Columns(SUNMatrix A)

Description This function returns the number of columns in the banded SUNMatrix.

F2003 Name This function is callable as FSUNBandMatrix_Columns when using the Fortran 2003 in-

terface module.

SUNBandMatrix_LowerBandwidth

Prototype sunindextype SUNBandMatrix_LowerBandwidth(SUNMatrix A)

Description This function returns the lower half-bandwidth of the banded SUNMatrix.

F2003 Name This function is callable as FSUNBandMatrix_LowerBandwidth when using the Fortran

2003 interface module.

SUNBandMatrix_UpperBandwidth

Prototype sunindextype SUNBandMatrix_UpperBandwidth(SUNMatrix A)

Description This function returns the upper half-bandwidth of the banded SUNMatrix.

F2003 Name This function is callable as FSUNBandMatrix_UpperBandwidth when using the Fortran

2003 interface module.

SUNBandMatrix_StoredUpperBandwidth

Prototype sunindextype SUNBandMatrix_StoredUpperBandwidth(SUNMatrix A)

Description This function returns the stored upper half-bandwidth of the banded SUNMatrix.

F2003 Name This function is callable as FSUNBandMatrix_StoredUpperBandwidth when using the

Fortran 2003 interface module.

SUNBandMatrix_LDim

Prototype sunindextype SUNBandMatrix_LDim(SUNMatrix A)

Description This function returns the length of the leading dimension of the banded SUNMatrix.

F2003 Name This function is callable as FSUNBandMatrix_LDim when using the Fortran 2003 interface

module.

SUNBandMatrix_Data

Prototype realtype* SUNBandMatrix_Data(SUNMatrix A)

Description This function returns a pointer to the data array for the banded SUNMatrix.

F2003 Name This function is callable as FSUNBandMatrix_Data when using the Fortran 2003 interface

module.

SUNBandMatrix_Cols

Prototype realtype** SUNBandMatrix_Cols(SUNMatrix A)

Description This function returns a pointer to the cols array for the banded SUNMatrix.

SUNBandMatrix_Column

Prototype realtype* SUNBandMatrix_Column(SUNMatrix A, sunindextype j)

Description This function returns a pointer to the diagonal entry of the j-th column of the banded SUNMatrix. The resulting pointer should be indexed over the range -mu to ml.

F2003 Name This function is callable as FSUNBandMatrix_Column when using the Fortran 2003 interface module.

Notes

- When looping over the components of a banded SUNMatrix A, the most efficient approaches are to:
 - First obtain the component array via A_data = SM_DATA_B(A) or A_data = SUNBandMatrix_Data(A) and then access A_data[i] within the loop.
 - First obtain the array of column pointers via A_cols = SM_COLS_B(A) or A_cols = SUNBandMatrix_Cols(A), and then access A_cols[j][i] within the loop.
 - Within a loop over the columns, access the column pointer via
 A_colj = SUNBandMatrix_Column(A,j) and then to access the entries within that column using SM_COLUMN_ELEMENT_B(A_colj,i,j).

All three of these are more efficient than using SM_ELEMENT_B(A,i,j) within a double loop.

• Within the SUNMatMatvec_Band routine, internal consistency checks are performed to ensure that the matrix is called with consistent NVECTOR implementations. These are currently limited to: NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS. As additional compatible vector implementations are added to SUNDIALS, these will be included within this compatibility check.

8.4.3 SUNMatrix Band Fortran interfaces

The SUNMATRIX_BAND module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The 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
```

data

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)

indexvals - pointer to a contiguous block of int variables (of length NNZ), containing the row indices (if CSC) or column indices (if CSR) of each nonzero matrix entry held in data

indexptrs - pointer to a contiguous block of int variables (of length NP+1). For CSC matrices each entry provides the index of the first column entry into the data and indexvals arrays, e.g. if indexptr[3]=7, then the first nonzero entry in the fourth column of the matrix is located in data[7], and is located in row indexvals[7] of the matrix. The last entry contains the total number of nonzero values in the matrix and hence points one past the end of the active data in the data and indexvals arrays. For CSR matrices, each entry provides the index of the first row entry into the data and indexvals arrays.

The following pointers are added to the SlsMat type for user convenience, to provide a more intuitive interface to the CSC and CSR sparse matrix data structures. They are set automatically when creating a sparse SUNMATRIX, based on the sparse matrix storage type.

```
rowvals - pointer to indexvals when sparsetype is CSC_MAT, otherwise set to NULL. colptrs - pointer to indexptrs when sparsetype is CSC_MAT, otherwise set to NULL. colvals - pointer to indexvals when sparsetype is CSR_MAT, otherwise set to NULL. rowptrs - pointer to indexptrs when sparsetype is CSR_MAT, otherwise set to NULL.
```

For example, the 5×4 CSC matrix

$$\begin{bmatrix} 0 & 3 & 1 & 0 \\ 3 & 0 & 0 & 2 \\ 0 & 7 & 0 & 0 \\ 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 5 \end{bmatrix}$$

could be stored in this structure as either

```
M = 5;
 N = 4;
 NNZ = 8;
  NP = N;
  data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0};
  sparsetype = CSC_MAT;
  indexvals = {1, 3, 0, 2, 0, 1, 3, 4};
  indexptrs = \{0, 2, 4, 5, 8\};
or
 M = 5;
 N = 4;
  NNZ = 10;
  NP = N;
  data = \{3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0, *, *\};
  sparsetype = CSC_MAT;
  indexvals = \{1, 3, 0, 2, 0, 1, 3, 4, *, *\};
  indexptrs = \{0, 2, 4, 5, 8\};
```

where the first has no unused space, and the second has additional storage (the entries marked with * may contain any values). Note in both cases that the final value in indexptrs is 8, indicating the total number of nonzero entries in the matrix.

Similarly, in CSR format, the same matrix could be stored as

```
M = 5;
N = 4;
NNZ = 8;
NP = M;
data = {3.0, 1.0, 3.0, 2.0, 7.0, 1.0, 9.0, 5.0};
sparsetype = CSR_MAT;
indexvals = {1, 2, 0, 3, 1, 0, 3, 3};
indexptrs = {0, 2, 4, 5, 7, 8};
```

The header file to include when using this module is sunmatrix/sunmatrix_sparse.h. The SUNMATRIX_SPARSE module is accessible from all SUNDIALS solvers without linking to the libsundials_sunmatrixsparse module library.

8.5.1 SUNMatrix_Sparse accessor macros

The following macros are provided to access the content of a SUNMATRIX_SPARSE matrix. The prefix SM_{-} in the names denotes that these macros are for SUNMatrix implementations, and the suffix $_{-}S$ denotes that these are specific to the sparse version.

• SM_CONTENT_S

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

The assignment $A_cont = SM_CONTENT_S(A)$ sets A_cont to be a pointer to the sparse SUNMatrix content structure.



Figure 8.2: Diagram of the storage for a compressed-sparse-column matrix. Here A is an $M \times N$ sparse matrix with storage for up to NNZ nonzero entries (the allocated length of both data and indexvals). The entries in indexvals may assume values from 0 to M-1, corresponding to the row index (zero-based) of each nonzero value. The entries in data contain the values of the nonzero entries, with the row i, column j entry of A (again, zero-based) denoted as A(i,j). The indexptrs array contains N+1 entries; the first N denote the starting index of each column within the indexvals and data arrays, while the final entry points one past the final nonzero entry. Here, although NNZ values are allocated, only nz are actually filled in; the greyed-out portions of data and indexvals indicate extra allocated space.

Implementation:

```
#define SM_CONTENT_S(A) ( (SUNMatrixContent_Sparse) (A->content) )
```

• SM_ROWS_S, SM_COLUMNS_S, SM_NNZ_S, SM_NP_S, and SM_SPARSETYPE_S

These macros give individual access to various lengths relevant to the content of a sparse SUNMatrix.

These may be used either to retrieve or to set these values. For example, the assignment A_rows = SM_ROWS_S(A) sets A_rows to be the number of rows in the matrix A. Similarly, the assignment SM_COLUMNS_S(A) = A_cols sets the number of columns in A to equal A_cols.

Implementation:

• 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:

8.5.2 SUNMatrix_Sparse functions

The SUNMATRIX_SPARSE module defines sparse implementations of all matrix operations listed in Section 8.1.1. Their names are obtained from those in Section 8.1.1 by appending the suffix _Sparse (e.g. SUNMatCopy_Sparse). All the standard matrix operations listed in Section 8.1.1 with the suffix _Sparse appended are callable via the FORTRAN 2003 interface by prepending an 'F' (e.g. FSUNMatCopy_Sparse).

The module SUNMATRIX_SPARSE provides the following additional user-callable routines:

SUNSparseMatrix

```
Prototype SUNMatrix SUNSparseMatrix(sunindextype M, sunindextype N, sunindextype NNZ, int sparsetype)
```

Description This function creates and allocates memory for a sparse SUNMatrix. Its arguments are the number of rows and columns of the matrix, M and N, the maximum number of nonzeros to be stored in the matrix, NNZ, and a flag sparsetype indicating whether to use CSR or CSC format (valid arguments are CSR_MAT or CSC_MAT).

F2003 Name This function is callable as FSUNSparseMatrix when using the Fortran 2003 interface module.

SUNSparseFromDenseMatrix

Prototype SUNMatrix SUNSparseFromDenseMatrix(SUNMatrix A, realtype droptol, int sparsetype);

Description This function creates a new sparse matrix from an existing dense matrix by copying all values with magnitude larger than droptol into the sparse matrix structure.

Requirements:

- A must have type SUNMATRIX_DENSE;
- droptol must be non-negative;
- sparsetype must be either CSC_MAT or CSR_MAT.

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

F2003 Name This function is callable as FSUNSparseFromDenseMatrix when using the Fortran 2003 interface module.

SUNSparseFromBandMatrix

Prototype SUNMatrix SUNSparseFromBandMatrix(SUNMatrix A, realtype droptol, int sparsetype);

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

Requirements:

- A must have type SUNMATRIX_BAND;
- droptol must be non-negative;
- sparsetype must be either CSC_MAT or CSR_MAT.

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

F2003 Name This function is callable as FSUNSparseFromBandMatrix when using the Fortran 2003 interface module.

SUNSparseMatrix_Realloc

Prototype int SUNSparseMatrix_Realloc(SUNMatrix A)

Description This function reallocates internal storage arrays in a sparse matrix so that the resulting sparse matrix has no wasted space (i.e. the space allocated for nonzero entries equals the actual number of nonzeros, indexptrs[NP]). Returns 0 on success and 1 on failure (e.g. if the input matrix is not sparse).

F2003 Name This function is callable as FSUNSparseMatrix_Realloc when using the Fortran 2003 interface module.

SUNSparseMatrix_Reallocate

Prototype int SUNSparseMatrix_Reallocate(SUNMatrix A, sunindextype NNZ)

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

F2003 Name This function is callable as FSUNSparseMatrix_Reallocate when using the Fortran 2003 interface module.

SUNSparseMatrix_Print

Prototype void SUNSparseMatrix_Print(SUNMatrix A, FILE* outfile)

Description This function prints the content of a sparse SUNMatrix to the output stream specified

by outfile. Note: stdout or stderr may be used as arguments for outfile to print

directly to standard output or standard error, respectively.

$SUNSparseMatrix_Rows$

Prototype sunindextype SUNSparseMatrix_Rows(SUNMatrix A)

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

F2003 Name This function is callable as FSUNSparseMatrix_Rows when using the Fortran 2003 inter-

face module.

${\tt SUNSparseMatrix_Columns}$

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.

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.

${\tt SUNSparseMatrix_IndexValues}$

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.

${\tt SUNSparseMatrix_IndexPointers}$

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, 30, 44, 45]. It is designed to be used with the SUNLINSOL_SUPERLUDIST linear solver discussed in Section 9.10. To this end, it defines the *content* field of SUNMatrix to be the following structure:

```
struct _SUNMatrixContent_SLUNRloc {
  booleantype    own_data;
  gridinfo_t    *grid;
  sunindextype    *row_to_proc;
  pdgsmv_comm_t    *gsmv_comm;
  SuperMatrix    *A_super;
  SuperMatrix    *ACS_super;
};
```

A more complete description of the this *content* field is given below:

own_data - a flag which indicates if the SUNMatrix is responsible for freeing A_super

grid - pointer to the SuperLU_DIST structure that stores the 2D process grid

row_to_proc - a mapping between the rows in the matrix and the process it resides on; will be NULL
until the SUNMatMatvecSetup routine is called

gsmv_comm - pointer to the SuperLU_DIST structure that stores the communication information needed for matrix-vector multiplication; will be NULL until the SUNMatMatvecSetup routine is called

A_super - pointer to the underlying SuperLU_DIST SuperMatrix with Stype = SLU_NR_loc, Dtype = SLU_D, Mtype = SLU_GE; must have the full diagonal present to be used with SUNMatScaleAddI routine

ACS_super - a column-sorted version of the matrix needed to perform matrix-vector multiplication; will be NULL until the routine SUNMatMatvecSetup routine is called

The header file to include when using this module is sunmatrix/sunmatrix_slunrloc.h. The installed module library to link to is libsundials_sunmatrixslunrloc.lib where .lib is typically .so for shared libraries and .a for static libraries.

8.6.1 SUNMatrix_SLUNRloc functions

The module SUNMATRIX_SLUNRLOC provides the following user-callable routines:

```
Call A = SUNMatrix_SLUNRloc(Asuper, grid);

Description The function SUNMatrix_SLUNRloc creates and allocates memory for a SUNMATRIX_SLUNRLOC object.

Arguments Asuper (SuperMatrix*) a fully-allocated SuperLU_DIST SuperMatrix that the SUNMatrix will wrap; must have Stype = SLU_NR_loc, Dtype = SLU_D, Mtype = SLU_GE to be compatible grid (gridinfo_t*) the initialized SuperLU_DIST 2D process grid structure

Return value a SUNMatrix object if Asuper is compatible else NULL
```

Notes

SUNMatrix_SLUNRloc_Print

Call SUNMatrix_SLUNRloc_Print(A, fp);

Description The function SUNMatrix_SLUNRloc_Print prints the underlying SuperMatrix content.

Arguments A (SUNMatrix) the matrix to print

fp (FILE) the file pointer used for printing

Return value void

Notes

SUNMatrix_SLUNRloc_SuperMatrix

Call Asuper = SUNMatrix_SLUNRloc_SuperMatrix(A);

Description The function SUNMatrix_SLUNRloc_SuperMatrix provides access to the underlying 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

Return value a SUNMatrix object if successful else NULL

SUNMatrix_cuSparse_Rows

Call M = SUNMatrix_cuSparse_Rows(A)

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

Arguments A (SUNMatrix)

Return value the number of rows in the sparse SUNMatrix

SUNMatrix_cuSparse_Columns

Call N = SUNMatrix_cuSparse_Columns(A)

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

Arguments A (SUNMatrix)

Return value the number of columns in the sparse SUNMatrix

SUNMatrix_cuSparse_NNZ

Call nnz = SUNMatrix_cuSparse_NNZ(A)

Description This function returns the number of nonzeros in the sparse SUNMatrix.

Arguments A (SUNMatrix)

Return value the number of nonzeros in the sparse SUNMatrix

SUNMatrix_cuSparse_SparseType

Call type = SUNMatrix_cuSparse_SparseType(A)

Description This function returns the sparsity format for the sparse SUNMatrix.

Arguments A (SUNMatrix)

Return value the SUNMAT_CUSPARSE_CSR or SUNMAT_CUSPARSE_BCSR sparsity formats

|SUNMatrix_cuSparse_IndexValues

Call colind = SUNMatrix_cuSparse_IndexValues(A)

Description This function returns a pointer to the index value array for the sparse SUNMatrix.

Arguments A (SUNMatrix)

Return value for the CSR format this is an array of the column indices for each nonzero entry. For the BCSR format this is an array of the column indices for each nonzero entry in the first block only.

|SUNMatrix_cuSparse_IndexPointers

Call rowptrs = SUNMatrix_cuSparse_IndexPointers(A)

Description This function returns a pointer to the index pointers array for the sparse SUNMatrix.

Arguments A (SUNMatrix)

Return value for the CSR format this is an array of the locations of the first entry of each row in the data and indexvalues arrays, for the BCSR format this is an array of the locations of each row in the data and indexvalues arrays in the first block only.

SUNMatrix_cuSparse_NumBlocks

Call nblocks = SUNMatrix_cuSparse_NumBlocks(A)

Description This function returns the number of blocks in the sparse SUNMatrix.

Arguments A (SUNMatrix)

Return value the number of matrix blocks

SUNMatrix_cuSparse_BlockRows

Call blockrows = SUNMatrix_cuSparse_BlockRows(A)

Description This function returns the number of rows of a block of the sparse SUNMatrix.

Arguments A (SUNMatrix)

Return value the number of rows of a block

|SUNMatrix_cuSparse_BlockColumns

Call 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)

* sizeof(realtype) bytes; the nonzero values will be copied from this array

onto the device

 $\verb|h_idxptrs| (int*) a pointer to an allocated array of at least (\verb|SUNMatrix_cuSparse_BlockDim(A)+1)| \\$

* 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)

 $\boldsymbol{*}$ $\boldsymbol{\texttt{sizeof}}$ (realtype) bytes; the nonzero values will be copied into this array

from the device

h_idxptrs (int*) a pointer to an allocated array of at least (SUNMatrix_cuSparse_BlockDim(A)+1)

* sizeof(int) bytes; the index pointers will be copied into this array from

the device

h_idxvals (int*) a pointer to an allocated array of at least SUNMatrix_cuSparse_BlockNNZ(A)

* sizeof(int) bytes; the index values will be copied into this array from

the device

Return value SUNMAT_SUCCESS if the copy operation(s) were successful, or a nonzero error code otherwise.

SUNMatrix_cuSparse_SetKernelExecPolicy

Call retval = SUNMatrix_cuSparse_SetKernelExecPolicy(A, exec_policy);

Description This function sets the execution policies which control the kernel parameters utilized

when launching the CUDA kernels. By default the matrix is setup to use a policy which tries to leverage the structure of the matrix. See section 7.9.2 for more information

about the SUNCudaExecPolicy class.

Arguments A (SUNMatrix)

exec_policy (SUNCudaExecPolicy*)

Return value SUNMAT_SUCCESS if the operation(s) were successful, or a nonzero error code otherwise.

Notes All matrices and vector used in a single instance of a SUNDIALS solver must use the

same CUDA stream, and the CUDA stream must be set prior to solver initialization.

|SUNMatrix_cuSparse_SetFixedPattern

Call retval = SUNMatrix_cuSparse_SetFixedPattern(A, yesno)

Description This function changes the behavior of the 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.

<u>!</u>

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 [52]. 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 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
- $5. \ {\tt SUNMatCopy_MagmaDense}$
- 6. SUNMatScaleAdd_MagmaDense
- 7. SUNMatScaleAddI_MagmaDense
- 8. SUNMatMatvecSetup_MagmaDense

9. SUNMatMatvec_MagmaDense

10. SUNMatSpace_MagmaDense

In addition, the SUNMATRIX_MAGMADENSE module defines the following implementation specific functions:

SUNMatrix_MagmaDense

Call A = SUNMatrix_MagmaDense(M, N, memtype, memhelper, queue)

Description This constructor function creates and allocates memory for an $M \times N$ Sunmatrix_magmadense

SUNMatrix.

Arguments M (sunindextype) the number of matrix rows

N (sunindextype) the number of matrix columns

memtype (SUNMemoryType) the type of memory to use for the matrix data; can be

SUNMEMTYPE_UVM or SUNMEMTYPE_DEVICE.

memhelper (SUNMemoryHelper) the memory helper used for allocating data

queue a cudaStream_t when using CUDA or a hipStream_t when using HIP

Return value A SUNMatrix object if successful else NULL.

SUNMatrix_MagmaDenseBlock

Call A = SUNMatrix_MagmaDenseBlock(nblocks, M_block, N_block, memtype, memhelper,

queue)

Description This constructor function creates and allocates memory for a SUNMATRIX_MAGMADENSE

SUNMatrix that is block diagonal with nblocks blocks of size $M \times N$.

Arguments nblocks (sunindextype) the number of matrix blocks

M_block (sunindextype) the number of matrix rows in each block
N_block (sunindextype) the number of matrix columns in each block

memtype (SUNMemoryType) the type of memory to use for the matrix data; can be

SUNMEMTYPE_UVM or SUNMEMTYPE_DEVICE.

memhelper (SUNMemoryHelper) the memory helper used for allocating data

queue a cudaStream_t when using CUDA or a hipStream_t when using HIP

Return value A SUNMatrix object if successful else NULL.

Notes The block diagonal format currently supports square matrices only.

SUNMatrix_MagmaDense_Rows

Call M = SUNMatrix_MagmaDense_Rows(A)

Description This function returns the rows dimension for the $M \times N$ SUNMatrix. For block diagonal

matrices, this is computed as $M_{\rm block} \times {\rm nblocks}$.

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 $N_{\text{block}} \times \text{nblocks}$.

Arguments A (SUNMatrix)

Return value The number of columns in the SUNMatrix.

SUNMatrix_MagmaDense_BlockRows

Call M = SUNMatrix_MagmaDense_BlockRows(A)

Description This function returns the number of rows in a block of the SUNMatrix.

Arguments A (SUNMatrix)

Return value The number of rows in a block of the SUNMatrix.

SUNMatrix_MagmaDense_BlockColumns

Call N = SUNMatrix_MagmaDense_BlockColumns(A)

Description This function returns the number of columns in a block of the SUNMatrix.

Arguments A (SUNMatrix)

Return value The number of columns in a block of the SUNMatrix.

SUNMatrix_MagmaDense_LData

Call ldata = SUNMatrix_MagmaDense_LData(A)

Description This function returns the length of the data array for the SUNMatrix.

Arguments A (SUNMatrix)

Return value The length of the data array for the SUNMatrix.

SUNMatrix_MagmaDense_NumBlocks

Call nblocks = SUNMatrix_MagmaDense_NumBlocks(A)

Description This function returns the number of blocks in the SUNMatrix.

Arguments A (SUNMatrix)

Return value The number of matrix blocks.

SUNMatrix_MagmaDense_Data

Call data = SUNMatrix_MagmaDense_Data(A)

Description This function returns the SUNMatrix data array.

Arguments A (SUNMatrix)

Return value An array of pointers to the data arrays for each block in the SUNMatrix.

SUNMatrix_MagmaDense_BlockData

Call data = SUNMatrix_MagmaDense_BlockData(A)

Description This function returns an array of pointers that point to the start of the data array for

each block.

Arguments A (SUNMatrix)

Return value An array of pointers to the data arrays for each block in the SUNMatrix.

SUNMatrix_MagmaDense_Block

Call data = SUNMatrix_MagmaDense_Block(A, k)

Description This function returns a pointer to the data for block k.

Arguments A (SUNMatrix)

Return value A pointer to the start of the data array for block k in the SUNMatrix.

Notes No bounds-checking is performed, k should be strictly less than nblocks.

SUNMatrix_MagmaDense_Column

Call data = SUNMatrix_MagmaDense_Column(A, j)

Description This function returns a pointer to the data for column j of the matrix.

Arguments A (SUNMatrix)

Return value A pointer to the start of the data array for column j of the SUNMatrix.

Notes No bounds-checking is performed, j should be strictly less than 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_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.

 $F2003\ Name\ FSUNLinSolInitialize$

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.

Toutilles

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 con-

taining 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	can	be	used	for
each.												

Linear Solver	Dense	Banded	Sparse	SLUNRloc	User
Interface	Matrix	Matrix	Matrix	Matrix	Supplied
Dense	√				✓
Band		✓			√
LapackDense	√				√
LapackBand		✓			√
KLU			✓		√
SuperLU_DIST				√	✓
SUPERLUMT			√		✓
User supplied	√	√	√	√	√

9.3 Implementing a custom SUNLinearSolver module

A particular implementation of the Sunlinsol module must:

- Specify the *content* field of the SUNLinearSolver object.
- Define and implement a minimal subset of the linear solver operations. See the section 9.4 to determine which SUNLINSOL operations are required for this SUNDIALS package.

Note that the names of these routines should be unique to that implementation in order to permit using more than one SUNLINSOL module (each with different SUNLinearSolver internal data representations) in the same code.

 Define and implement user-callable constructor and destructor routines to create and free a SUNLinearSolver with the new content field and with ops pointing to the new linear solver operations.

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

To aid in the creation of custom SUNLINSOL modules the generic SUNLINSOL module provides the utility functions SUNLinSolNewEmpty and SUNLinSolFreeEmpty. When used in custom SUNLINSOL constructors the function SUNLinSolNewEmpty will ease the introduction of any new optional linear solver operations to the SUNLINSOL API by ensuring only required operations need to be set.

SUNLinSolNewEmpty

Call LS = SUNLinSolNewEmpty();

Description The function SUNLinSolNewEmpty allocates a new generic SUNLINSOL object and initializes its content pointer and the function pointers in the operations structure to NULL.

Arguments None

Return value This function returns a SUNLinearSolver object. If an error occurs when allocating the object, then this routine will return NULL.

F2003 Name FSUNLinSolNewEmpty

SUNLinSolFreeEmpty

Call SUNLinSolFreeEmpty(LS);

Description This routine frees the generic SUNLinSolFreeEmpty object, under the assumption that

any implementation-specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the ops pointer is NULL,

and, if it is not, it will free it as well.

Arguments LS (SUNLinearSolver)

Return value None

F2003 Name FSUNLinSolFreeEmpty

Additionally, a SUNLINSOL implementation may do the following:

- Define and implement additional user-callable "set" routines acting on the SUNLinearSolver, e.g., for setting various configuration options to tune the linear solver to a particular problem.
- Provide additional user-callable "get" routines acting on the SUNLinearSolver object, e.g., for returning various solve statistics.

9.3.1 Intended use cases

The SUNLINSOL (and SUNMATRIX) APIs are designed to require a minimal set of routines to ease interfacing with custom or third-party linear solver libraries. External solvers provide similar routines with the necessary functionality and thus will require minimal effort to wrap within custom SUNMATRIX and SUNLINSOL implementations. Sections 8.2 and 9.4 include a list of the required set of routines that compatible SUNMATRIX and SUNLINSOL implementations must provide. As SUNDIALS packages utilize generic SUNLINSOL modules allowing for user-supplied SUNLinearSolver implementations, there exists a wide range of possible linear solver combinations. Some intended use cases for both the SUNDIALS-provided and user-supplied SUNLINSOL modules are discussed in the following sections.

Direct linear solvers

Direct linear solver modules require a matrix and compute an 'exact' solution to the linear system defined by the matrix. Multiple matrix formats and associated direct linear solvers are supplied with SUNDIALS through different SUNMATRIX and SUNLINSOL implementations. SUNDIALS packages strive to amortize the high cost of matrix construction by reusing matrix information for multiple nonlinear iterations. As a result, each package's linear solver interface recomputes Jacobian information as infrequently as possible.

Alternative matrix storage formats and compatible linear solvers that are not currently provided by, or interfaced with, SUNDIALS can leverage this infrastructure with minimal effort. To do so, a user must implement custom SUNMATRIX and SUNLINSOL wrappers for the desired matrix format and/or linear solver following the APIs described in Chapters 8 and 9. This user-supplied SUNLINSOL module must then self-identify as having SUNLINEARSOLVER_DIRECT type.

Matrix-free iterative linear solvers

Matrix-free iterative linear solver modules do not require a matrix and compute an inexact solution to the linear system defined by the package-supplied ATimes routine. SUNDIALS supplies multiple scaled, preconditioned iterative linear solver (spils) SUNLINSOL modules that support scaling to allow users to handle non-dimensionalization (as best as possible) within each SUNDIALS package and retain variables and define equations as desired in their applications. For linear solvers that do not support left/right scaling, the tolerance supplied to the linear solver is adjusted to compensate (see section 9.4.2 for more details); however, this use case may be non-optimal and cannot handle situations where the magnitudes of different solution components or equations vary dramatically within a single problem.

To utilize alternative linear solvers that are not currently provided by, or interfaced with, 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 CVODE SUNLinearSolver interface

Table 9.3 below lists the SUNLINSOL module linear solver functions used within the CVLS interface. As with the SUNMATRIX module, we emphasize that the CVODE user does not need to know detailed usage of linear solver functions by the CVODE code modules in order to use CVODE. 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. SUNLinSolNumIters is only used to accumulate overall iterative linear solver statistics. If it is not implemented by the SUNLINSOL module, then CVLS will consider all solves as requiring zero iterations.
- 2. Although CVLS does not call SUNLinSolLastFlag directly, this routine is available for users to query linear solver issues directly.
- 3. Although CVLS does not call SUNLinSolFree directly, this routine should be available for users to call when cleaning up from a simulation.

	DIRECT	ITERATIVE	MATRIX_ITERATIVE
SUNLinSolGetType	√	√	✓
SUNLinSolSetATimes	†	√	†
SUNLinSolSetPreconditioner	†	†	†
SUNLinSolSetScalingVectors	†	†	†
SUNLinSolInitialize	√	√	✓
SUNLinSolSetup	√	√	✓
SUNLinSolSolve	√	√	√
1 SUNLinSolNumIters		†	†
2 SUNLinSolLastFlag			
³ SUNLinSolFree			
SUNLinSolSpace	†	†	†

Table 9.3: List of linear solver function usage in the CVLS interface

Since there are a wide range of potential SUNLINSOL use cases, the following subsections describe some details of the CVLS interface, in the case that interested users wish to develop custom SUNLINSOL modules.

9.4.1 Lagged matrix information

If the SUNLINSOL object self-identifies as having type SUNLINEARSOLVER_DIRECT or SUNLINEARSOLVER_MATRIX_ITERATIVE, then the SUNLINSOL object solves a linear system defined by a SUNMATRIX object. CVLS 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 with $M = (I - \gamma J)$, and the actual linear system

$$\bar{M}\bar{x} = b \quad \Leftrightarrow \quad (I - \bar{\gamma}J)\bar{x} = b.$$

Since CVLS updates the SUNMATRIX object infrequently, it is likely that $\gamma \neq \bar{\gamma}$, and in turn $M \neq \bar{M}$. When using a BDF method, after calling the SUNLINSOL-provided SUNLinSolSolve routine, we test whether $\gamma/\bar{\gamma} \neq 1$, and if this is the case we scale the solution \bar{x} to correct the linear system solution x via

$$x = \frac{2}{1 + \gamma/\bar{\gamma}}\bar{x}.\tag{9.3}$$

The motivation for this selection of the scaling factor $c = 2/(1 + \gamma/\bar{\gamma})$ is discussed in detail in [13, 35]. 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 CVLS 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 CVLS 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 CVODE 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)):

$$\begin{split} &\left\|\tilde{b} - \tilde{A}\tilde{x}\right\|_2 < \text{tol} \\ \Leftrightarrow &\left\|SP_1^{-1}b - SP_1^{-1}Ax\right\|_2 < \text{tol} \\ \Leftrightarrow &\left.\sum_{i=0}^{n-1} \left[W_i \left(P_1^{-1}(b-Ax)\right)_i\right]^2 < \text{tol}^2 \\ \Leftrightarrow &\left.W_{mean}^2 \sum_{i=0}^{n-1} \left[\left(P_1^{-1}(b-Ax)\right)_i\right]^2 < \text{tol}^2 \\ \Leftrightarrow &\left.\sum_{i=0}^{n-1} \left[\left(P_1^{-1}(b-Ax)\right)_i\right]^2 < \left(\frac{\text{tol}}{W_{mean}}\right)^2 \\ \Leftrightarrow &\left.\|P_1^{-1}(b-Ax)\right\|_2 < \frac{\text{tol}}{W_{mean}} \end{split}$$

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 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.

SUNLinSol_Band
DOMPTHDOT_Dand

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.
- ullet SUNLinSolSetup_Band this performs the LU factorization.
- SUNLinSolSolve_Band this uses the LU factors and pivots array to perform the solve.
- SUNLinSolLastFlag_Band
- SUNLinSolSpace_Band this only returns information for the storage within the solver object, i.e. storage for N, last_flag, and pivots.
- SUNLinSolFree_Band

All of the listed operations are callable via the FORTRAN 2003 interface module by prepending an 'F' to the function name.

9.6.3 SUNLinearSolver_Band Fortran interfaces

The SUNLINSOL_BAND module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The fsunlinsol_band_mod FORTRAN module defines interfaces to all SUNLINSOL_BAND C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function SUNLinSol_Band is interfaced as FSUNLinSol_Band.

The FORTRAN 2003 SUNLINSOL_BAND interface module can be accessed with the use statement, i.e. use fsunlinsol_band_mod, and linking to the library libsundials_fsunlinsolband_mod.lib in addition to the C library. For details on where the library and module file fsunlinsol_band_mod.mod are installed see Appendix A. We note that the module is accessible from the FORTRAN 2003 SUNDIALS integrators without separately linking to the libsundials_fsunlinsolband_mod library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL_BAND module also includes a Fortran-callable function for creating a SUNLinearSolver object.

FSUNBANDLINSOLINIT

Call FSUNBANDLINSOLINIT(code, ier)

 $\label{thm:constraint} \textbf{Description} \quad \textbf{The function } \textbf{FSUNBANDLINSOLINIT} \ \text{can be called for Fortran programs to create a band}$

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 \mathtt{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_LapackDer	ıse
---------------------	-----

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.

 $Deprecated\ Name\ For\ backward\ compatibility,\ the\ wrapper\ function\ {\tt SUNLapackDense}\ with\ idential$

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.
- ullet SUNLinSolSetup_LapackDense this calls either DGETRF or SGETRF to perform the LU factorization.
- SUNLinSolSolve_LapackDense this calls either DGETRS or SGETRS to use the *LU* factors and pivots array to perform the solve.
- SUNLinSolLastFlag_LapackDense
- SUNLinSolSpace_LapackDense this only returns information for the storage within the solver object, i.e. storage for N, last_flag, and pivots.
- SUNLinSolFree_LapackDense

9.7.3 SUNLinearSolver_LapackDense Fortran interfaces

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL_LAPACKDENSE module also includes a Fortran-callable function for creating a SUNLinearSolver object.

FSUNLAPACKDENSEINIT

Call 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.



9.8.1SUNLinearSolver_LapackBand description

This solver is constructed to perform the following operations:

- The "setup" call performs a LU factorization with partial (row) pivoting, PA = LU, where P is a permutation matrix, L is a lower triangular matrix with 1's on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX_BAND object A, with pivoting information encoding P stored in the pivots array.
- The "solve" call performs pivoting and forward and backward substitution using the stored pivots array and the LU factors held in the SUNMATRIX_BAND object.
- A must be allocated to accommodate the increase in upper bandwidth that occurs during factorization. More precisely, if A is a band matrix with upper bandwidth mu and lower bandwidth m1, then the upper triangular factor U can have upper bandwidth as big as smu = MIN(N-1, mu+ml). The lower triangular factor L has lower bandwidth ml.



9.8.2SUNLinearSolver_LapackBand functions

The SUNLINSOL_LAPACKBAND module provides the following user-callable constructor for creating a SUNLinearSolver object.

CIIMI inCol	LapackBand
PONTIUPOT.	LapackBand

Notes

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 SUNLinearSolver object. If either A or y are incompatible then this

routine will return NULL.

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 ε is the double-precision unit roundoff), then a new factorization is performed.
- The module includes the routine SUNKLUReInit, that can be called by the user to force a full or partial refactorization at the next "setup" call.
- The "solve" call performs pivoting and forward and backward substitution using the stored KLU data structures. We note that in this solve KLU operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

9.9.2 SUNLinearSolver_KLU functions

The SUNLINSOL_KLU module provides the following user-callable constructor for creating a SUNLinearSolver object.

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.

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

Notes

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.

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)

 $\label{thm:constraint} \textbf{Description} \quad \textbf{The function } \textbf{FSUNKLUSETORDERING } \ \text{can be called for Fortran programs to change } \ \text{the function } \textbf{FSUNKLUSETORDERING } \ \text{can be called for Fortran programs } \ \text{to change } \ \text{the function } \ \text{Fortran } \ \text{for function } \ \text{function } \ \text{for function } \ \text{function } \ \text{function$

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, 30, 44, 45]. 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 structure that contains the 2D process grid.

Arguments LS (SUNLinearSolver) the SUNLINSOL_SUPERLUDIST object

Return value gridinfo_t*

Notes

SUNLinSol_SuperLUDIST_GetLUstruct

Call LUstruct_t *lu = SUNLinSol_SuperLUDIST_GetLUstruct(LS);

Description The function SUNLinSol_SuperLUDIST_GetLUstruct returns the SuperLU_DIST struc-

ture that contains the distributed L and U factors.

Arguments LS (SUNLinearSolver) the SUNLINSOL_SUPERLUDIST object

Return value LUstruct_t*

Notes

${\tt SUNLinSol_SuperLUDIST_GetSuperLUOptions}$

Call superlu_dist_options_t *opts = SUNLinSol_SuperLUDIST_GetSuperLUOptions(LS);

Description The function SUNLinSol_SuperLUDIST_GetSuperLUOptions returns the SuperLU_DIST

structure that contains the options which control how the linear system is factorized

and solved.

Arguments LS (SUNLinearSolver) the SUNLINSOL_SUPERLUDIST object

Return value superlu_dist_options_t*

Notes

SUNLinSol_SuperLUDIST_GetScalePermstruct

Call ScalePermstruct_t *sp = SUNLinSol_SuperLUDIST_GetScalePermstruct(LS);

Description The function SUNLinSol_SuperLUDIST_GetScalePermstruct returns the SuperLU_DIST

structure that contains the vectors that describe the transformations done to the matrix,

A.

Arguments LS (SUNLinearSolver) the SUNLINSOL_SUPERLUDIST object

Return value ScalePermstruct_t*

Notes

|SUNLinSol_SuperLUDIST_GetSOLVEstruct

Call SOLVEstruct_t *solve = SUNLinSol_SuperLUDIST_GetSOLVEstruct(LS);

Description The function SUNLinSol_SuperLUDIST_GetSOLVEstruct returns the SuperLU_DIST 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,

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, 43, 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 = SUNLin	Sol_SuperLUMT(y, A, num_threads);	
Description		SUNLinSol_SuperLUMT creates and allocates memory for a '-based SUNLinearSolver object.	
Arguments	y A	$({\tt N_Vector})$ a template for cloning vectors needed within the solver $({\tt SUNMatrix})$ a Sunmatrix_sparse matrix template for cloning matrices needed within the solver	
	numthreads	(\mathtt{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 ${\tt SUNLinearSolver}$ object. If either A or y are incompatible then this routine will return ${\tt NULL}$.		
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.

tionally when using ARKORE with a non identity ma

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)

 $\label{thm:constraint} \textbf{Description} \quad \textbf{The function } \textbf{FSUNMASSSUPERLUMTINIT} \ \ \textbf{can be called for Fortran programs to create a}$

SuperLU_MT-based SUNLinearSolver object for mass matrix linear systems.

Arguments num_threads (int*) desired number of threads (OpenMP or Pthreads, depending on

how superlumt was installed) to use during the factorization steps.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes This routine must be called after both the NVECTOR and SUNMATRIX mass-matrix

objects have been initialized.

The SUNLinSol_SuperLUMTSetOrdering routine also supports Fortran interfaces for the system and mass matrix solvers:

FSUNSUPERLUMTSETORDERING

Call FSUNSUPERLUMTSETORDERING(code, ordering, ier)

Description The function FSUNSUPERLUMTSETORDERING can be called for Fortran programs to update

the ordering algorithm in a SUNLINSOL_SUPERLUMT object.

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

for KINSOL, and 4 for ARKODE).

ordering (int*) a flag indicating the ordering algorithm, options are:

0 natural ordering

1 minimal degree ordering on A^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 rou-

tine.

FSUNMASSUPERLUMTSETORDERING

Call FSUNMASSUPERLUMTSETORDERING(ordering, ier)

Description The function FSUNMASSUPERLUMTSETORDERING can be called for Fortran programs to

update the ordering algorithm in a SUNLINSOL_SUPERLUMT object for mass matrix linear

systems.

Arguments ordering (int*) a flag indicating the ordering algorithm, options are:

0 natural ordering

```
1 minimal degree ordering on A^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.

SUNLinearSolver_SuperLUMT content

options

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.
```

9.12The SUNLinearSolver_cuSolverSp_batchQR implementation

The SUNLinearSolver_cuSolverSp_batchQR implementation of the SUNLINSOL API is designed to be used with the SUNMATRIX_CUSPARSE matrix, and the NVECTOR_CUDA vector. The header file to include when using this module is sunlinsol/sunlinsol_cusolversp_batchqr.h. The installed library to link to is libsundials_sunlinsolcusolversp. lib where . lib is typically .so for shared libraries and .a for static libraries.

The SUNLinearSolver_cuSolverSp_batchQR module is experimental and subject to change.



9.12.1 SUNLinearSolver_cuSolverSp_batchQR description

The SUNLinearSolver_cuSolverSp_batchQR implementation provides an interface to the batched sparse QR factorization method provided by the NVIDIA cuSOLVER library [6]. The module is designed for solving block diagonal linear systems of the form

$$\begin{bmatrix} \mathbf{A_1} & 0 & \cdots & 0 \\ 0 & \mathbf{A_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A_n} \end{bmatrix} x_j = b_j$$

where all block matrices $\mathbf{A_j}$ share the same 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 [52].

The header file to include when using this module is $sunlinsol_magmadense.h$. The installed library to link to is $libsundials_sunlinsol_magmadense.lib$ where .lib is typically .so for shared libraries and .a for static libraries.



The SUNLinearSolver_MagmaDense module is experimental and subject to change.

9.13.1 SUNLinearSolver_MagmaDense description

The SUNLinearSolver_MagmaDense implementation provides an interface to the dense LU and dense batched LU methods in the MAGMA linear algebra library [4]. The batched LU methods are leveraged when solving block diagonal linear systems of the form

$$\begin{bmatrix} \mathbf{A_0} & 0 & \cdots & 0 \\ 0 & \mathbf{A_1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A_{n-1}} \end{bmatrix} x_j = b_j.$$

9.13.2 SUNLinearSolver_MagmaDense functions

The SUNLinearSolver_MagmaDense module defines implementations of all "direct" linear solver operations listed in Sections 9.1.1-9.1.3:

- SUNLinSolGetType_MagmaDense
- SUNLinSolInitialize_MagmaDense
- SUNLinSolSetup_MagmaDense
- SUNLinSolSolve_MagmaDense
- SUNLinSolLastFlag_MagmaDense
- SUNLinSolFree_MagmaDense

In addition, the module provides the following user-callable routines:

SUNLinSol_MagmaDense

Call LS = SUNLinSol_MagmaDense(y, A);

Description The function SUNLinSol_MagmaDense creates and allocates memory for a SUNLINSOL object.

Arguments y (N_Vector) a vector for checking compatibility with the solver

A (SUNMatrix) a SUNMATRIX_MAGMADENSE matrix for checking compatibility with the solver

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

Notes This routine analyzes the input matrix and vector to determine the linear system size and to assess compatibility with the solver.

```
Call SUNLinSol_MagmaDense_SetAsync(SUNLinearSolver LS, booleantype onoff);

Description The function SUNLinSol_MagmaDense_SetAsync can be used to toggle the linear solver between asynchronous and synchronous modes. In asynchronous mode, SUNLinearSolver operations are asynchronous with respect to the host. In synchronous mode, the host and GPU device are synchronized prior to the operation returning.

Arguments LS (SUNLinearSolver) a SUNLinSol_MagmaDense object onoff (booleantype) set to 0 for synchronous mode, or 1 for asynchronous mode

Return value None

Notes The default is asynchronous mode.
```

9.13.3 SUNLinearSolver_MagmaDense content

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

```
struct _SUNLinearSolverContent_MagmaDense {
  int
                  last_flag;
  booleantype
                  async;
  sunindextype
                  N;
  SUNMemory
                  pivots;
  SUNMemory
                  pivotsarr;
  SUNMemory
                  dpivotsarr;
  SUNMemory
                  infoarr;
  SUNMemory
                  rhsarr;
  SUNMemoryHelper memhelp;
  magma_queue_t
};
```

9.14 The SUNLinearSolver_SPGMR implementation

This section describes the SUNLINSOL implementation of the SPGMR (Scaled, Preconditioned, Generalized Minimum Residual [50]) 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 ≤ 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

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 maxl are the same as for the C function

SUNLinSol_SPGMR.

The SUNLinSol_SPGMRSetPrecType, SUNLinSol_SPGMRSetGSType and

SUNLinSol_SPGMRSetMaxRestarts routines also support Fortran interfaces for the system and mass matrix solvers.

FSUNSPGMRSETGSTYPE

Call FSUNSPGMRSETGSTYPE(code, gstype, ier)

Description The function FSUNSPGMRSETGSTYPE can be called for Fortran programs to change the

Gram-Schmidt 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 [49]) 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 ≤ 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_SPFGMR 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

. .

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 \mathtt{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.

Notes

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.

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.

Notes See SUNLinSol_SPFGMRSetMaxRestarts for complete further documentation of this routine.

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 $z_1, \ldots, z_{\max 1+1}$, stored in Z[0], ..., Z[max1]. 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*max1 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 .

zcor - 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),

 g_j ,

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 [53]) 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 ≤ 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	Tvpe(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 ≤ 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 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 [29]) 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 ≤ 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.

${\tt SUNLinSol_SPTFQMRSetPrecType}$

Call retval = SUNLinSol_SPTFQMRSetPrecType(LS, pretype);

Description The function SUNLinSol_SPTFQMRSetPrecType updates the type of preconditioning

to use in the SUNLINSOL_SPTFQMR object.

Arguments LS (SUNLinearSolver) the SUNLINSOL_SPTFQMR object to update

 ${\tt pretype}$ (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 ≤ 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

9.17.3 SUNLinearSolver_SPTFQMR Fortran interfaces

The SUNLINSOL_SPFGMR module provides a FORTRAN 2003 module as well as FORTRAN 77 style interface functions for use from FORTRAN applications.

FORTRAN 2003 interface module

The fsunlinsol_sptfqmr_mod FORTRAN module defines interfaces to all SUNLINSOL_SPFGMR C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading 'F'. For example, the function SUNLinSol_SPTFQMR is interfaced as FSUNLinSol_SPTFQMR.

The FORTRAN 2003 SUNLINSOL_SPFGMR interface module can be accessed with the use statement, i.e. use fsunlinsol_sptfqmr_mod, and linking to the library libsundials_fsunlinsolsptfqmr_mod. lib in addition to the C library. For details on where the library and module file

fsunlinsol_sptfqmr_mod.mod are installed see Appendix A. We note that the module is accessible from the Fortran 2003 sundials integrators without separately linking to the libsundials_fsunlinsolsptfqmr_mod library.

FORTRAN 77 interface functions

For solvers that include a FORTRAN 77 interface module, the SUNLINSOL_SPTFQMR module also includes a Fortran-callable function for creating a SUNLinearSolver object.

FSUNSPTFQMRINIT

Call FSUNSPTFQMRINIT(code, pretype, maxl, ier)

Description The function FSUNSPTFQMRINIT can be called for Fortran programs to create a SUNLIN-

SOL_SPTFQMR object.

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

for KINSOL, and 4 for ARKODE).

pretype (int*) flag indicating desired preconditioning type

maxl (int*) flag indicating number of iterations to allow

Return value ier is a return completion flag equal to 0 for a success return and -1 otherwise. See

printed message for details in case of failure.

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

Allowable values for pretype and 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 [31]) 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 δ .

This solver is constructed to perform the following operations:

- During construction all NVECTOR solver data is allocated, with vectors cloned from a template NVECTOR that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLINSOL_PCG to supply the ATimes, PSetup, and Psolve function pointers and s scaling vector.
- In the "initialize" call, the solver parameters are checked for validity.
- In the "setup" call, any non-NULL PSetup function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic PSetup function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call the PCG iteration is performed. This will include scaling and preconditioning if those options have been supplied.

9.18.2 SUNLinearSolver_PCG functions

The SUNLINSOL_PCG module provides the following user-callable constructor for creating a SUNLinearSolver object.

CITAIT	C-7	DOG
SUM	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 y (N_Vector) a template for cloning vectors needed within the solver

pretype (int) flag indicating whether to use preconditioning. Since the PCG algorithm is designed to only support symmetric preconditioning, then any of the pretype inputs PREC_LEFT (1), PREC_RIGHT (2), or PREC_BOTH (3) will result in use of the symmetric preconditioner; any other integer input

will result in the default (no preconditioning).

maxl (int) the number of linear iterations to allow; values ≤ 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.

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_PCGSetMax1

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 ≤ 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 SUNPCGSetMaxl 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).

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)

Description The function FSUNPCGSETPRECTYPE can be called for Fortran programs to change the

type of preconditioning to use.

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

for KINSOL, and 4 for ARKODE).

pretype (int*) flag indicating the type of preconditioning to use.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol_PCGSetPrecType for complete further documentation of this routine.

FSUNMASSPCGSETPRECTYPE

Call FSUNMASSPCGSETPRECTYPE(pretype, ier)

Description The function FSUNMASSPCGSETPRECTYPE can be called for Fortran programs to change

the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to FSUNPCGSETPRECTYPE above, except that code is not

needed since mass matrix linear systems only arise in ARKODE.

Return value ier is a int return completion flag equal to 0 for a success return and -1 otherwise.

See printed message for details in case of failure.

Notes See SUNLinSol_PCGSetPrecType for complete further documentation of this routine.

FSUNPCGSETMAXL

Call FSUNPCGSETMAXL(code, maxl, ier)

Description The function FSUNPCGSETMAXL can be called for Fortran programs to change the 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

struct _SUNLinearSolverContent_PCG {

The SUNLINSOL_PCG 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 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:
             - number of PCG iterations to allow (default is 5),
maxl
             - 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.

ycor (N_Vector) on input the initial guess for the correction to the predicted state (zero) and on output the final correction to the predicted state. See section 10.2 for more detail on the nonlinear system formulation.

w (N_Vector) the solution error weight vector used for computing weighted

error norms.

tol (realtype) the requested solution tolerance in the weighted root-mean-

squared norm.

callLSetup (booleantype) a flag indicating that the integrator recommends for the

linear solver setup function to be called.

mem (void *) 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 (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

Call retval = SUNNonlinSolFree(NLS);

Description The optional function SUNNonlinSolFree frees any memory allocated by the nonlinear

solver.

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) = 0 for SUNNONLINEARSOLVER_ROOTFIND modules or G(y) in G(y) = y for

SUNNONLINEARSOLVER_FIXEDPOINT modules.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object.

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 $A = \frac{\partial F}{\partial u}$ 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. Sunnonlinear implementations that do not require solving this system or do not use Sunlinear solvers may

set this operation to NULL.

F2003 Name FSUNNonlinSolSetLSolveFn

SUNNonlinSolSetConvTestFn

Call retval = SUNNonlinSolSetConvTestFn(NLS, CTestFn, ctest_data);

 ${\tt Description} \quad \text{The } \textit{optional } \textbf{function } \textbf{SUNNonlinSolSetConvTestFn} \text{ is used to provide the nonlinear } \textbf{SunnonlinSolSetConvTestFn} \text{ is used to provide the nonlinear } \textbf{SunnonlinSolSetConvTestFn} \text{ is used to provide the nonlinear } \textbf{SunnonlinSolSetConvTestFn} \text{ is used to provide the nonlinear } \textbf{SunnonlinSolSetConvTestFn} \text{ is used to provide the nonlinear } \textbf{SunnonlinSolSetConvTestFn} \text{ is used to provide the nonlinear } \textbf{SunnonlinSolSetConvTestFn} \text{ is used to provide the nonlinear } \textbf{SunnonlinSolSetConvTestFn} \text{ is used to provide the nonlinear } \textbf{SunnonlinSolSetConvTestFn} \text{ is used to provide the nonlinear } \textbf{SunnonlinSolSetConvTestFn} \text{ is used to provide the nonlinear } \textbf{SunnonlinSolSetConvTestFn} \text{ is used to provide the nonlinear } \textbf{SunnonlinSolSetConvTestFn} \text{ is used to provide the nonlinear } \textbf{SunnonlinSolSetConvTestFn} \text{ is used to provide the nonlinear } \textbf{SunnonlinSolSetConvTestFn} \text{ is used to provide the nonlinear } \textbf{SunnonlinSolSetConvTestFn} \text{ is used to provide the nonlinear } \textbf{Sunnonlinear } \textbf$

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);

The optional function SUNNonlinSolGetNumConvFails returns the number of nonlinear Description

> 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

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);

These functions evaluate the nonlinear system F(y) for SUNNONLINEARSOLVER_ROOTFIND Purpose 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

typedef int (*SUNNonlinSolLSetupFn)(booleantype jbad, booleantype* jcur, Definition void* mem);

Purpose These functions are wrappers to the SUNDIALS integrator's function for setting up linear

solves with SUNLINSOL modules.

jbad is an input indicating whether the nonlinear solver believes that A has gone stale Arguments

(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

Definition typedef int (*SUNNonlinSolConvTestFn)(SUNNonlinearSolver NLS, N_Vector ycor, N_Vector del, realtype tol,

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:

SUN_NLS_SUCCESS the iteration is converged.

SUN_NLS_CONTINUE the iteration has not converged, keep iterating.

SUN_NLS_CONV_RECVR the iteration appears to be diverging, try to recover.

Notes

The tolerance passed to this routine by SUNDIALS integrators is the tolerance in a weighted root-mean-squared norm with error weight vector ewt. SUNNONLINSOL modules utilizing their own convergence criteria may ignore these functions.

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 non-linear 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
- 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 NULL,

and, if it is not, it will free it as well.

Arguments NLS (SUNNonlinearSolver)

Return value None

F2003 Name FSUNNonlinSolFreeEmpty

10.2 CVODE SUNNonlinear Solver interface

As discussed in Chapter 2 each integration step requires the (approximate) solution of a nonlinear system. This system can be formulated as the rootfinding problem

$$F(y^n) \equiv y^n - h_n \beta_{n,0} f(t_n, y^n) - a_n = 0, \qquad (10.1)$$

or as the fixed-point problem

$$G(y^n) \equiv h_n \beta_{n,0} f(t_n, y^n) + a_n = y^n,$$
 (10.2)

where $a_n \equiv \sum_{i>0} (\alpha_{n,i} y^{n-i} + h_n \beta_{n,i} \dot{y}^{n-i}).$

Rather than solving the above nonlinear systems for the new state y^n CVODE reformulates the above problems to solve for the correction y_{cor} to the predicted new state y_{pred} so that $y^n = y_{pred} + y_{cor}$. The nonlinear systems rewritten in terms of y_{cor} are

$$F(y_{cor}) \equiv y_{cor} - \gamma f(t_n, y^n) - \tilde{a}_n = 0, \qquad (10.3)$$

for the rootfinding problem and

$$G(y_{cor}) \equiv \gamma f(t_n, y^n) + \tilde{a}_n = y_{cor}. \tag{10.4}$$

for the fixed-point problem.

The nonlinear system functions provided by CVODE to the nonlinear solver module internally update the current value of the new state based on the input correction vector i.e., $y^n = y_{pred} + y_{cor}$. The updated vector y^n is used when calling the ODE right-hand side function and when setting up linear solves (e.g., updating the Jacobian or preconditioner).

CVODE 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 value of γ to compute Jacobian data.

CVodeGetCurrentGamma

Call flag = CVodeGetCurrentGamma(cvode_mem, &gamma);

Description The function CVodeGetCurrentGamma returns the current value of the scalar γ .

Arguments cvode_mem (void *) pointer to the CVODE memory block.

gamma (realtype *) the current value of the scalar γ appearing in the Newton equation $M = I - \gamma J$.

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

F2003 Name FCVodeGetCurrentGamma

CVodeGetCurrentState

Call flag = CVodeGetCurrentState(cvode_mem, &y);

Description The function CVodeGetCurrentState returns the current state vector. When called within the computation of a step (i.e., during a nonlinear solve) this is $y^n = y_{pred} + y_{cor}$. Otherwise this is the current internal solution vector y(t). In either case the

corresponding solution time can be obtained from CVodeGetCurrentTime.

Arguments cvode_mem (void *) pointer to the CVODE memory block.

y (N_Vector *) pointer that is set to the current state vector

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

F2003 Name FCVodeGetCurrentState

${\tt CVodeGetNonlinearSystemData}$

Call flag = CVodeGetNonlinearSystemData(cvode_mem, &tn, &ypred, &yn, &fn, &gamma, &rl1, &zn1, &user_data);

Description The function CVodeGetNonlinearSystemData returns all internal data required to construct the current nonlinear system (10.3) or (10.4).

Arguments cvode_mem (void *) pointer to the CVODE memory block.

tn (realtype*) current value of the independent variable t_n .

ypred (N_Vector*) predicted state vector y_{pred} at t_n . This vector must not be

changed.

yn $(N_{\text{-}Vector*})$ state vector y^n . This vector may be not current and may need

to be filled (see the note below).

fn (N_Vector*) the right-hand side function evaluated at the current time and state, $f(t_n, y^n)$. This vector may be not current and may need to be filled

(see the note below).

gamma (realtype*) current value of γ .

rl1 (realtype*) a scaling factor used to compute $\tilde{a}_n = rl1 * zn1$.

zn1 (N_Vector*) a vector used to compute $\tilde{a}_n = rl1 * zn1$.

user_data (void**) pointer to the user-defined data structures

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

CV_SUCCESS The optional output values have been successfully set.

CV_MEM_NULL The cvode_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 CVodeGetNonlinearSystemData inside the nonlinear system function to access the requisite data for evaluting the nonlinear system function of their choosing. Additionly, if the SUNNonlinearSolver object (existing or custom) leverages the SUNNonlinSolLSetupFn and/or SUNNonlinSolLSolveFn functions supplied by CVODE (through calls to SUNNonlinSolSetLSetupFn and SUNNonlinSolSetLSolveFn respectively) the vectors yn and fn 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.,

$$yn = y_{pred} + y_{cor},$$

$$fn = f(t_n, y^n),$$

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 yn and fn are only current when CVodeGetNonlinearSystemData is called after an evaluation of the nonlinear system function.

F2003 Name FCVodeGetNonlinearSystemData

CVodeComputeState

Call flag = CVodeComputeState(cvode_mem, ycor, yn);

Description The function computes the current y(t) vector based on stored prediction and the given

correction vector from the nonlinear solver i.e., $y^n = y_{pred} + y_{cor}$.

Arguments cvode_mem - (void *) pointer to the CVODE memory block

ycor - (N_Vector) the correction yn - (N_Vector) the output vector

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

 $F2003\ \mathrm{Name}\ FCVodeComputeState}$

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

$$F(y) = 0 ag{10.5}$$

given an initial guess $y^{(0)}$, Newton's method computes a series of approximate solutions

$$y^{(m+1)} = y^{(m)} + \delta^{(m+1)} \tag{10.6}$$

where m is the Newton iteration index, and the Newton update $\delta^{(m+1)}$ is the solution of the linear system

$$A(y^{(m)})\delta^{(m+1)} = -F(y^{(m)}), (10.7)$$

in which A is the Jacobian matrix

$$A \equiv \partial F/\partial y \,. \tag{10.8}$$

Depending on the linear solver used, the SUNNONLINSOL_NEWTON module will employ either a Modified Newton method, or an Inexact Newton method [12, 15, 23, 25, 42]. When used with a direct linear solver, the Jacobian matrix A is held constant during the Newton iteration, resulting in a Modified Newton method. With a matrix-free iterative linear solver, the iteration is an Inexact Newton method.

In both cases, calls to the integrator-supplied SUNNonlinSollSetupFn function are made infrequently to amortize the increased cost of matrix operations (updating A and its factorization within direct linear solvers, or updating the preconditioner within iterative linear solvers). Specifically, SUNNONLINSOL_NEWTON will call the SUNNonlinSollSetupFn function in two instances:

- (a) when requested by the integrator (the input callLSetSetup is SUNTRUE) before attempting the Newton iteration, or
- (b) when reattempting the nonlinear solve after a recoverable failure occurs in the Newton iteration with stale Jacobian information (jcur is SUNFALSE). In this case, SUNNONLINSOL_NEWTON will set jbad to SUNTRUE before calling the SUNNonlinSollSetupFn function.

Whether the Jacobian matrix A is fully or partially updated depends on logic unique to each integrator-supplied SUNNonlinSolSetupFn routine. We refer to the discussion of nonlinear solver strategies provided in 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);

Description The function SUNNonlinSolSetInfoFile_Newton sets the output file where all informa-

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

SUNNonlinear Solver object for use with SUNDIALS integrators to solve nonlinear systems of the form F(x) = 0 with Newton's method

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:
             - the function for evaluating the nonlinear system,
Sys
LSetup
             - the package-supplied function for setting up the linear solver,
LSolve
             - the package-supplied function for performing a linear solve,
CTest
             - the function for checking convergence of the Newton iteration,
delta
             - the Newton iteration update vector,
             - the Jacobian status (SUNTRUE = current, SUNFALSE = stale),
icur
             - the current number of iterations in the solve attempt,
curiter
             - the maximum number of Newton iterations allowed in a solve,
maxiters
             - the total number of nonlinear iterations across all solves,
niters
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 file
             - the file where all informative (non-error) messages will be directed
```

10.4 The SUNNonlinearSolver_FixedPoint implementation

This section describes the SUNNONLINSOL implementation of a fixed point (functional) iteration with optional Anderson acceleration. To access the SUNNONLINSOL_FIXEDPOINT module, include the header file sunnonlinsol/sunnonlinsol_fixedpoint.h. We note that the SUNNONLINSOL_FIXEDPOINT module is accessible from SUNDIALS integrators without separately linking to the libsundials_sunnonlinsolfixedpoint module library.

10.4.1 SUNNonlinearSolver_FixedPoint description

To find the solution to

$$G(y) = y (10.9)$$

given an initial guess $y^{(0)}$, the fixed point iteration computes a series of approximate solutions

$$y^{(n+1)} = G(y^{(n)}) (10.10)$$

where n is the iteration index. The convergence of this iteration may be accelerated using Anderson's method [10, 54, 28, 46]. With Anderson acceleration using subspace size m, the series of approximate solutions can be formulated as the linear combination

$$y^{(n+1)} = \beta \sum_{i=0}^{m_n} \alpha_i^{(n)} G(y^{(n-m_n+i)}) + (1-\beta) \sum_{i=0}^{m_n} \alpha_i^{(n)} y_{n-m_n+i}$$
 (10.11)

where $m_n = \min\{m, n\}$, β is a damping parameter ($\beta = 1.0$ i.e., no damping by default) and the factors

$$\alpha^{(n)} = (\alpha_0^{(n)}, \dots, \alpha_{m_n}^{(n)}) \tag{10.12}$$

solve the minimization problem $\min_{\alpha} \|F_n \alpha^T\|_2$ under the constraint that $\sum_{i=0}^{m_n} \alpha_i = 1$ where

$$F_n = (f_{n-m_n}, \dots, f_n) \tag{10.13}$$

with $f_i = G(y^{(i)}) - y^{(i)}$. Due to this constraint, in the limit of m = 0 the accelerated fixed point iteration formula (10.11) simplifies to the standard fixed point iteration (10.10).

Following the recommendations made in [54], the SUNNONLINSOL_FIXEDPOINT implementation computes the series of approximate solutions as

$$y^{(n+1)} = G(y^{(n)}) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta g_{n-m_n+i} - (1-\beta)(f(y^{(n)}) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta f_{n-m_n+i})$$
(10.14)

with $\Delta g_i = G(y^{(i+1)}) - G(y^{(i)})$ and where the factors

$$\gamma^{(n)} = (\gamma_0^{(n)}, \dots, \gamma_{m_n-1}^{(n)}) \tag{10.15}$$

solve the unconstrained minimization problem $\min_{\gamma} \|f_n - \Delta F_n \gamma^T\|_2$ where

$$\Delta F_n = (\Delta f_{n-m_n}, \dots, \Delta f_{n-1}), \tag{10.16}$$

with $\Delta f_i = f_{i+1} - f_i$. The least-squares problem is solved by applying a QR factorization to $\Delta F_n = Q_n R_n$ and solving $R_n \gamma = Q_n^T f_n$.

The acceleration subspace size m is required when constructing the SUNNONLINSOL_FIXEDPOINT object. The default maximum number of iterations and the stopping criteria for the fixed point iteration are supplied by the SUNDIALS integrator when SUNNONLINSOL_FIXEDPOINT is attached to it. Both the maximum number of iterations and the convergence test function may be modified by the user by calling SUNNonlinSolSetMaxIters and SUNNonlinSolSetConvTestFn functions after attaching the SUNNONLINSOL_FIXEDPOINT object to the integrator.

10.4.2 SUNNonlinearSolver_FixedPoint functions

The SUNNONLINSOL_FIXEDPOINT module provides the following constructors for creating a SUNNonlinearSolver object.

SUNNonlinSol_FixedPoint

Call NLS = SUNNonlinSol_FixedPoint(y, m);

Description The function SUNNonlinSol_FixedPoint creates a SUNNonlinearSolver object for use

with SUNDIALS integrators to solve nonlinear systems of the form G(y) = y.

Arguments y (N_Vector) a template for cloning vectors needed within the solver

m (int) the number of acceleration vectors to use

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_FixedPoint

SUNNonlinSol_FixedPointSens

Call NLS = SUNNonlinSol_FixedPointSens(count, y, m);

Description The function SUNNonlinSol_FixedPointSens creates a SUNNonlinearSolver object for

use with SUNDIALS sensitivity enabled integrators (CVODES and IDAS) to solve nonlinear

systems of the form G(y) = y.

Arguments count (int) the number of vectors in the nonlinear solve. When integrating a system

• Ns+1 if using a *simultaneous* corrector approach.

containing ${\tt Ns}$ sensitivities the value of ${\tt count}$ is:

• Ns if using a *staggered* corrector approach.

y (N_Vector) a template for cloning vectors needed within the solver.

m (int) the number of acceleration vectors to use.

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_FixedPointSens

Since the accelerated fixed point iteration (10.10) does not require the setup or solution of any linear systems, the SUNNONLINSOL_FIXEDPOINT module implements all of the functions defined in sections 10.1.1 - 10.1.3 except for the SUNNonlinSolSetup, SUNNonlinSolSetUsetupFn, and

SUNNonlinSolSetLSolveFn functions, that are set to NULL. The SUNNONLINSOL_FIXEDPOINT functions have the same names as those defined by the generic SUNNONLINSOL API with <code>FixedPoint</code> appended to the function name. Unless using the SUNNONLINSOL_FIXEDPOINT 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_FIXEDPOINT-specific implementations.

The SUNNONLINSOL_FIXEDPOINT module also defines the following additional user-callable functions.

${\tt SUNNonlinSolGetSysFn_FixedPoint}$

Call retval = SUNNonlinSolGetSysFn_FixedPoint(NLS, SysFn);

Description The function SUNNonlinSolGetSysFn_FixedPoint returns the fixed-point 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 fixed-point function in a

custom convergence test function for the SUNNONLINSOL_FIXEDPOINT module. We note that SUNNONLINSOL_FIXEDPOINT will not leverage the results from any user calls to

SysFn.

F2003 Name FSUNNonlinSolGetSysFn_FixedPoint

SUNNonlinSolSetDamping_FixedPoint

Call retval = SUNNonlinSolSetDamping_FixedPoint(NLS, beta);

Description The function SUNNonlinSolSetDamping_FixedPoint sets the damping parameter β to

use with Anderson acceleration. By default damping is disabled i.e., $\beta = 1.0$.

Arguments NLS (SUNNonlinearSolver) a SUNNONLINSOL object

beta (realtype) the damping parameter $0 < beta \le 1$.

Return value The return value retval (of type int) is zero for a successful call, SUN_NLS_MEM_NULL if

NLS is NULL, or SUN_NLS_ILL_INPUT if beta is negative.

Notes A beta value should be greater than zero and less than one if damping is to be used.

A value of one or more will disable damping.

10.4.3 SUNNonlinear Solver Fixed Point Fortran interfaces

The SUNNONLINSOL_FIXEDPOINT 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_fixedpoint_mod FORTRAN module defines interfaces to all SUNNONLINSOL_FIXEDPOINT 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_FixedPoint is interfaced as FSUNNonlinSol_FixedPoint.

The FORTRAN 2003 SUNNONLINSOL_FIXEDPOINT interface module can be accessed with the use statement, i.e. use fsunnonlinsol_fixedpoint_mod, and linking to the library

libsundials_fsunnonlinsolfixedpoint_mod.lib in addition to the C library. For details on where the library and module file fsunnonlinsol_fixedpoint_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_fsunnonlinsolfixedpoint_mod library.

FORTRAN 77 interface functions

For SUNDIALS integrators that include a FORTRAN 77 interface, the SUNNONLINSOL_FIXEDPOINT module also includes a Fortran-callable function for creating a SUNNonlinearSolver object.

FSUNFIXEDPOINTINIT

Call FSUNFIXEDPOINTINIT(code, m, ier);

Description The function FSUNFIXEDPOINTINIT can be called for Fortran programs to create a

SUNNonlinearSolver object for use with SUNDIALS integrators to solve nonlinear sys-

tems of the form G(y) = y.

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

for ARKODE).

m (int*) is an integer input specifying the number of acceleration vectors.

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.

SUNNonlinSolSetInfoFile_FixedPoint

Call retval = SUNNonlinSolSetInfoFile_FixedPoint(NLS, info_file);

Description The function SUNNonlinSolSetInfoFile_FixedPoint sets the output file where all in-

formative (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_FixedPoint

|SUNNonlinSolSetPrintLevel_FixedPoint

Call retval = SUNNonlinSolSetPrintLevel_FixedPoint(NLS, print_level);

Description The function SUNNonlinSolSetPrintLevel_FixedPoint specifies the level of verbosity of the output.

or 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.

 $F2003 \text{ Name } FSUNNonlinSolSetPrintLevel_FixedPoint$

10.4.4 SUNNonlinearSolver_FixedPoint content

The sunnonlinearSolver as the following structure:

```
realtype *cvals;
  N_Vector *df;
  N_Vector *dg;
  N_Vector *q;
  N_Vector *Xvecs;
  N_Vector
            yprev;
  N_Vector
             gy;
  N_Vector
            fold;
  N_Vector gold;
  N_Vector
             delta;
  int
              curiter;
  int
              maxiters;
  long int niters;
             nconvfails;
  long int
  void*
              ctest_data;
              print_level;
  int
  FILE*
              info_file;
};
The following entries of the content field are always allocated:
             - function for evaluating the nonlinear system,
Sys
             - function for checking convergence of the fixed point iteration,
CTest
             - N_Vector used to store previous fixed-point iterate,
yprev
             - N_Vector used to store G(y) in fixed-point algorithm,
gу
delta
             - N_Vector used to store difference between successive fixed-point iterates,
             - the current number of iterations in the solve attempt,
curiter
             - the maximum number of fixed-point iterations allowed in a solve,
maxiters
             - the total number of nonlinear iterations across all solves,
niters
nconvfails - the total number of nonlinear convergence failures across all solves,
ctest_data - the data pointer passed to the convergence test function, and
             - number of acceleration vectors,
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
If Anderson acceleration is requested (i.e., m > 0 in the call to SUNNonlinSol.FixedPoint), then the
following items are also allocated within the content field:
          - index array used in acceleration algorithm (length m),
damping - a flag indicating if damping is enabled,
beta
          - the damping parameter,
          - small matrix used in acceleration algorithm (length m*m),
gamma
          - small vector used in acceleration algorithm (length m),
cvals
          - small vector used in acceleration algorithm (length m+1),
          - array of N_Vectors used in acceleration algorithm (length m),
df
          - array of N_Vectors used in acceleration algorithm (length m),
dg
          - array of N_Vectors used in acceleration algorithm (length m),
q
Xvecs
          - N_Vector pointer array used in acceleration algorithm (length m+1),
fold
          - N_Vector used in acceleration algorithm, and
          - N_Vector used in acceleration algorithm.
gold
```

10.5 The SUNNonlinear Solver_Petsc SNES 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/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.

10.5.1 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 [11].



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 SUNNON-LINSOL_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.

10.5.2 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 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);

Description The function 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

Return value The return value retval (of type int) should be zero for a successful call, and a negative value for a failure.

|SUNNonlinSolGetSysFn_PetscSNES

Call retval = SUNNonlinSolGetSysFn_PetscSNES(NLS, SysFn);

Description The function SUNNonlinSolGetSysFn_PetscSNES returns the residual function that de-

fines 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.

10.5.3 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,
{\tt petsc\_last\_err} \ - \ {\rm 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
```

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
```

```
Call retval = SUNMemoryHelper_Alloc(helper, *memptr, mem_size, mem_type);

Description Allocates a SUNMemory object whose ptr field is allocated for mem_size bytes and is of type mem_type. The new object will have ownership of ptr and will be deallocated when SUNMemoryHelper_Dealloc is called.

Arguments helper (SUNMemoryHelper) the SUNMemoryHelper object memptr (SUNMemory*) pointer to the allocated SUNMemory mem_size (size_t) the size in bytes of the ptr mem_type (SUNMemoryType) the SUNMemoryType of the ptr

Return value An int flag indicating success (zero) or failure (non-zero).
```

SUNMemoryHelper_Dealloc

```
Call retval = SUNMemoryHelper_Dealloc(helper, mem);

Description Deallocates the mem->ptr field if it is owned by mem, and then deallocates the mem object.

Arguments helper (SUNMemoryHelper) the SUNMemoryHelper object mem (SUNMemory) the SUNMemory object
```

Return value An int flag indicating success (zero) or failure (non-zero).

Return value An int flag indicating success (zero) or failure (non-zero).

```
SUNMemoryHelper_Copy
```

```
Call retval = SUNMemoryHelper_Copy(helper, dst, src, mem_size);

Description Synchronously copies mem_size bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The helper object should use the memory types of dst and src to determine the appropriate transfer type necessary.

Arguments helper (SUNMemoryHelper) the SUNMemoryHelper object dst (SUNMemory) the destination memory to copy to src (SUNMemory) the source memory to copy from mem_size (size_t) the number of bytes to copy
```

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

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.

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:

${\tt SUNMemoryHelper_Alloc_Cuda}$

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.

```
(SUNMemoryHelper) the SUNMemoryHelper object
Return value helper
             dst
                       (SUNMemory) the destination memory to copy to
                       (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); Asynchronously 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 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

(void *) the cudaStream_t handle for the stream that the copy will be perctx

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

Return value A SUNMemoryHelper object if successful, or NULL if not.

SUNMemoryHelper API functions 11.3.1

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);
```

Allocates a SUNMemory object whose ptr field is allocated for mem_size bytes and is Description 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)
- \bullet 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 build dir enter the ccmake command and point to the solver dir:

% ccmake ../solverdir

The default configuration screen is shown in Figure A.1.

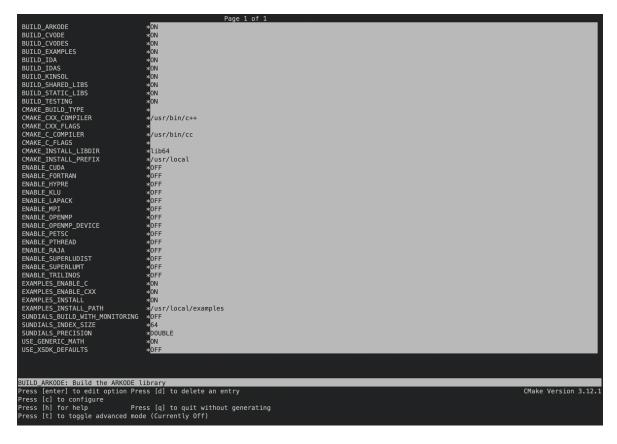


Figure A.1: Default configuration screen. Note: Initial screen is empty. To get this default configuration, press 'c' repeatedly (accepting default values denoted with asterisk) until the 'g' option is available.

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

 ${\tt CMAKE_CXX_COMPILER - C^{++} \ compiler}$

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:

 ${\tt CMAKE_Fortran_FLAGS_DEBUG~-} \ \, {\tt Flags~used~by~the~Fortran~compiler~during~debug~builds}$

Default: -g

 ${\tt CMAKE_Fortran_FLAGS_MINSIZEREL~-Flags~used~by~the~Fortran~compiler~during~release~minsize~builds}$

Default: -Os

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

 ${\tt EXAMPLES_ENABLE_CUDA}$ - Build the SUNDIALS CUDA examples

Default: OFF

Note: You need to enable CUDA support to build these examples.

 $EXAMPLES_ENABLE_CXX$ - Build the SUNDIALS C^{++} examples

Default: OFF unless ENABLE_TRILINGS 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=0FF. CMAKE_C_FLAGS if USE_XSDK_DEFAULTS=0N.

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

 $\begin{array}{c} {\tt USE_GENERIC_MATH\ -\ Use\ generic\ (stdc)\ math\ libraries} \\ {\tt Default:\ ON} \end{array}$

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)
```

Table A.1: SUNDIALS libraries and header files

	Libraries	ALS libraries and header files
SHARED		
	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	$libsundials_nvecparallel.lib$
11,12010101110112222	210101100	libsundials_fnvecparallel.a
		libsundials_fnvecparallel_mod.lib
	Header files	nvector/nvector_parallel.h
	Module	fnvector_parallel_mod.mod
	files	invoctor_paranot_mod.mod
NVECTOR_MANYVECTOR	Libraries	libsundials_nvecmanyvector.lib
NVECTOR_MANTVECTOR	Libraries	libsundials_nvecmanyvector_mod.lib
	Header files	nvector/nvector_manyvector.h
	Module Module	,
	files	fnvector_manyvector_mod.mod
NUECTOD MDIMANNATOCTOR	Libraries	liboundiala nyaamnimanyyystä lih
NVECTOR_MPIMANYVECTOR	Libraries	libsundials_nvecmpimanyvector.lib
	TT 1. C1	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 IIII delli
		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	${ m edpoint_mod.} lib$	
	Header files	sunnonlinsol/sunnonlinsol_fi	xedpoint.h	
	Module	fsunnonlinsol_fixedpoint_mod.mod		
	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		
15.1	Listaires	libsundials_fida.a		
		libsundials_fida_mod.lib		
	Header files	ida/ida.h	ida/ida_impl.h	
	TICKGCI IIICS	ida/ida_direct.h	ida/ida_ls.h	
		ida/ida_spils.h	ida/ida_bbdpre.h	
	Module	fida_mod.mod	ida, ida_bbapic.ii	
	files	nga_mou.mou		
	11169		continued on next page	
			сопынией он нехт раде	

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

CVODE 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 CVODE input constants

CVODE main solver module					
CV_ADAMS	1	Adams-Moulton linear multistep method.			
CV_BDF	2	BDF linear multistep method.			
CV_NORMAL	1	Solver returns at specified output time.			
CV_ONE_STEP 2 Solver returns after each successful step.					
	I	terative linear solver modules			
PREC_NONE	0	No preconditioning			
PREC_LEFT	1	Preconditioning on the left only.			
PREC_RIGHT	2	Preconditioning on the right only.			
PREC_BOTH	3	Preconditioning on both the left and the right.			
MODIFIED_GS	1	Use modified Gram-Schmidt procedure.			
CLASSICAL_GS	2	Use classical Gram-Schmidt procedure.			

B.2 CVODE output constants

CVODE main solver module			
CV_SUCCESS	0	Successful function return.	
CV_TSTOP_RETURN	1	CVode succeeded by reaching the specified stopping point.	
CV_ROOT_RETURN	2	CVode succeeded and found one or more roots.	
CV_WARNING	99	CVode succeeded but an unusual situation occurred.	
CV_TOO_MUCH_WORK	-1	The solver took mxstep internal steps but could not reach tout.	
CV_TOO_MUCH_ACC	-2	The solver could not satisfy the accuracy demanded by the user	
		for some internal step.	

392 CVODE Constants

CV_ERR_FAILURE	-3	Error test failures occurred too many times during one internal
		time step or minimum step size was reached.
CV_CONV_FAILURE	-4	Convergence test failures occurred too many times during one
		internal time step or minimum step size was reached.
CV_LINIT_FAIL	-5	The linear solver's initialization function failed.
CV_LSETUP_FAIL	-6	The linear solver's setup function failed in an unrecoverable
		manner.
CV_LSOLVE_FAIL	-7	The linear solver's solve function failed in an unrecoverable
		manner.
$CV_RHSFUNC_FAIL$	-8	The right-hand side function failed in an unrecoverable manner.
CV_FIRST_RHSFUNC_ERR	-9	The right-hand side function failed at the first call.
CV_REPTD_RHSFUNC_ERR	-10	The right-hand side function had repetead recoverable errors.
CV_UNREC_RHSFUNC_ERR	-11	The right-hand side function had a recoverable error, but no
		recovery is possible.
CV_RTFUNC_FAIL	-12	The rootfinding function failed in an unrecoverable manner.
CV_NLS_INIT_FAIL	-13	The nonlinear solver's init routine failed.
CV_NLS_SETUP_FAIL	-14	The nonlinear solver's setup routine failed.
CV_CONSTR_FAIL	-15	The inequality constraints were violated and the solver was un-
		able to recover.
CV_NLS_FAIL	-16	The nonlinear solver failed in an unrecoverable manner.
CV_MEM_FAIL	-20	A memory allocation failed.
CV_MEM_NULL	-21	The cvode_mem argument was NULL.
CV_ILL_INPUT	-22	One of the function inputs is illegal.
CV_NO_MALLOC	-23	The CVODE memory block was not allocated by a call to
		CVodeMalloc.
CV_BAD_K	-24	The derivative order k is larger than the order used.
CV_BAD_T	-25	The time t is outside the last step taken.
CV_BAD_DKY	-26	The output derivative vector is NULL.
CV_TOO_CLOSE	-27	The output and initial times are too close to each other.
CV_VECTOROP_ERR	-28	A vector operation failed.
CV_PROJ_MEM_NULL	-29	The projection memory was NULL.
CV_PROJFUNC_FAIL	-30	The projection function failed in an unrecoverable manner.
CV_REPTD_PROJFUNC_ERR	-31	The projection function had repeated recoverable errors.
		CVLS linear solver interface
CVI S SUCCESS	0	Successful function return

0	Successful function return.
-1	The cvode_mem argument was NULL.
-2	The CVLS linear solver has not been initialized.
-3	The CVLS solver is not compatible with the current NVECTOR
	module.
-4	A memory allocation request failed.
-5	The preconditioner module has not been initialized.
-6	The Jacobian function failed in an unrecoverable manner.
-7	The Jacobian function had a recoverable error.
	-1 -2 -3 -4 -5 -6

CVDIAG_RHSFUNC_UNRECVR

CVDIAG_RHSFUNC_RECVR

-6

-7

CVLS_SUNMAT_FAIL CVLS_SUNLS_FAIL	-8 -9	An error occurred with the current SUNMATRIX module. An error occurred with the current SUNLINSOL module.
		CVDIAG linear solver module
CVDIAG_SUCCESS	0	Successful function return.
CVDIAG_MEM_NULL	-1	The cvode_mem argument was NULL.
CVDIAG_LMEM_NULL	-2	The CVDIAG linear solver has not been initialized.
${\tt CVDIAG_ILL_INPUT}$	-3	The CVDIAG solver is not compatible with the current NVECTOR
		module.
CVDIAG_MEM_FAIL	-4	A memory allocation request failed.
CVDIAG_INV_FAIL	-5	A diagonal element of the Jacobian was 0.

The right-hand side function failed in an unrecoverable manner.

The right-hand side function had a recoverable error.

Appendix C

SUNDIALS Release History

Table C.1: Release History

Da	ate	SUNDIALS	ARKODE	CVODE	CVODES	IDA	IDAS	KINSOL
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	_	_	$1.0.0^{1}$	_	_	_	_
¹ CVODE written, ² PVODE written, ³ CVODE and PVODE combined, ⁴ IDA written, ⁵ 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] D. G. Anderson. Iterative procedures for nonlinear integral equations. J. Assoc. Comput. Machinery, 12:547–560, 1965.
- [11] 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.
- [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, G. D. Byrne, and A. C. Hindmarsh. VODE, a Variable-Coefficient ODE Solver. SIAM J. Sci. Stat. Comput., 10:1038–1051, 1989.
- [14] P. N. Brown and A. C. Hindmarsh. Reduced Storage Matrix Methods in Stiff ODE Systems. *J. Appl. Math. & Comp.*, 31:49–91, 1989.
- [15] P. N. Brown and Y. Saad. Hybrid Krylov Methods for Nonlinear Systems of Equations. SIAM J. Sci. Stat. Comput., 11:450–481, 1990.
- [16] 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.
- [17] G. D. Byrne and A. C. Hindmarsh. A Polyalgorithm for the Numerical Solution of Ordinary Differential Equations. *ACM Trans. Math. Softw.*, 1:71–96, 1975.

398 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 User Guide. Technical Report UCRL-MA-118618, LLNL, September 1994.
- [21] S. D. Cohen and A. C. Hindmarsh. CVODE, a Stiff/Nonstiff ODE Solver in C. Computers in Physics, 10(2):138–143, 1996.
- [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] Edda Eich. Convergence results for a coordinate projection method applied to mechanical systems with algebraic constraints. SIAM Journal on Numerical Analysis, 30(5):1467–1482, 1993.
- [28] H. Fang and Y. Saad. Two classes of secant methods for nonlinear acceleration. *Numer. Linear Algebra Appl.*, 16:197–221, 2009.
- [29] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. SIAM J. Sci. Comp., 14:470–482, 1993.
- [30] 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.
- [31] 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.
- [32] K. L. Hiebert and L. F. Shampine. Implicitly Defined Output Points for Solutions of ODEs. Technical Report SAND80-0180, Sandia National Laboratories, February 1980.
- [33] A. C. Hindmarsh. Detecting Stability Barriers in BDF Solvers. In J.R. Cash and I. Gladwell, editor, *Computational Ordinary Differential Equations*, pages 87–96, Oxford, 1992. Oxford University Press.
- [34] A. C. Hindmarsh. Avoiding BDF Stability Barriers in the MOL Solution of Advection-Dominated Problems. Appl. Num. Math., 17:311–318, 1995.
- [35] A. C. Hindmarsh. The PVODE and IDA Algorithms. Technical Report UCRL-ID-141558, LLNL, December 2000.
- [36] 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.

BIBLIOGRAPHY 399

[37] A. C. Hindmarsh, R. Serban, and A. Collier. Example Programs for IDA v5.7.0. Technical Report UCRL-SM-208113, LLNL, 2021.

- [38] A. C. Hindmarsh, R. Serban, and D. R. Reynolds. Example Programs for CVODE v5.7.0. Technical report, LLNL, 2021. UCRL-SM-208110.
- [39] 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.
- [40] K. R. Jackson and R. Sacks-Davis. An Alternative Implementation of Variable Step-Size Multistep Formulas for Stiff ODEs. ACM Trans. Math. Softw., 6:295–318, 1980.
- [41] Seth R. Johnson, Andrey Prokopenko, and Katherine J. Evans. Automated fortran-c++ bindings for large-scale scientific applications. arXiv:1904.02546 [cs], 2019.
- [42] C. T. Kelley. Iterative Methods for Solving Linear and Nonlinear Equations. SIAM, Philadelphia, 1995.
- [43] X. S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.
- [44] 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.
- [45] 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.
- [46] P. A. Lott, H. F. Walker, C. S. Woodward, and U. M. Yang. An accelerated Picard method for nonlinear systems related to variably saturated flow. Adv. Wat. Resour., 38:92–101, 2012.
- [47] K. Radhakrishnan and A. C. Hindmarsh. Description and Use of LSODE, the Livermore Solver for Ordinary Differential Equations. Technical Report UCRL-ID-113855, LLNL, march 1994.
- [48] Daniel R. Reynolds. Example Programs for ARKODE v4.7.0. Technical report, Southern Methodist University, 2021.
- [49] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. SIAM J. Sci. Comput., 14(2):461–469, 1993.
- [50] 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.
- [51] LF Shampine. Conservation laws and the numerical solution of odes, ii. Computers & Mathematics with Applications, 38(2):61–72, 1999.
- [52] 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.
- [53] 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.
- [54] H. F. Walker and P. Ni. Anderson acceleration for fixed-point iterations. SIAM Jour. Num. Anal., 49(4):1715–1735, 2011.

Index

Adams method, 19	usage, $89-90$
	user-callable functions, 90
BDF method, 19	CVBandPrecGetNumRhsEvals, 91
BIG_REAL, 32, 130, 137	CVBandPrecGetWorkSpace, 90
booleantype, 32	CVBandPrecInit, 90
	CVBBDPRE preconditioner
CONSTR_VEC, 114	description, 91–92
CV_ADAMS, 38, 79	optional output, 95–96
CV_BAD_DKY, 63	usage, 93–94
CV_BAD_K, 63	user-callable functions, 94–95
CV_BAD_T, 63	user-supplied functions, 92–93
CV_BDF, 38, 79	CVBBDPrecGetNumGfnEvals, 96
CV_CONSTR_FAIL, 45	CVBBDPrecGetWorkSpace, 96
CV_CONV_FAILURE, 45	CVBBDPrecInit, 94
CV_ERR_FAILURE, 45	CVBBDPrecReInit, 95
CV_FIRST_RHSFUNC_ERR, 80	CVDIAG linear solver
CV_FIRST_RHSFUNC_FAIL, 45	Jacobian approximation used by, 42
CV_ILL_INPUT, 38, 39, 43, 45, 49–51, 53, 54, 60,	selection of, 42
79	CVDIAG linear solver interface
CV_LINIT_FAIL, 45	memory requirements, 77
CV_LSETUP_FAIL, 45, 83, 85, 92, 93	optional output, 77–79
CV_LSOLVE_FAIL, 45	use in FCVODE, 112
CV_MEM_FAIL, 38, 52, 70, 71	CVDiag, 36, 41, 42
CV_MEM_NULL, 38, 39, 43, 45, 46, 48–54, 60–63, 65–	CVDIAG_ILL_INPUT, 42
72, 79	CVDIAG_LMEM_NULL, 78
CV_NO_MALLOC, 39, 45, 79	CVDIAG_MEM_FAIL, 42
CV_NORMAL, 44	CVDIAG_MEM_NULL, 42, 78
CV DDO I MEM NULL 61 62 72	CVDIAG_SUCCESS, 42, 78
CV_PROJ_MEM_NULL, 61, 62, 72 CV_REPTD_RHSFUNC_ERR, 45	CVDiagGetLastFlag, 78
CV_RHSFUNC_FAIL, 45, 80	CVDiagGetNumRhsEvals, 78
CV_ROOT_RETURN, 44	CVDiagGetReturnFlagName, 79
CV_RTFUNC_FAIL, 45, 82	CVDiagGetWorkSpace, 78
CV_SUCCESS, 38, 39, 43, 44, 46, 48–54, 60–63, 65–	CVDlsGetLastFlag, 77
72, 79	CVDlsGetNumJacEvals, 73
CV_TOO_CLOSE, 45	CVDlsGetNumRhsEvals, 74
CV_TOO_MUCH_ACC, 45	CVDlsGetReturnFlagName, 77
CV_TOO_MUCH_WORK, 45	CVDlsGetWorkspace, 73
CV_TSTOP_RETURN, 44	CVDlsJacFn, 84
CV_UNREC_RHSFUNC_ERR, 45, 80	CVDlsSetJacFn, 55
CV_WARNING, 81	CVDlsSetLinearSolver, 42
CVBANDPRE preconditioner	CVErrHandlerFn, 80
description, 89	CVEwtFn, 81
optional output, 90–91	CVLS generic linear solver
1	0

SUNLINSOL compatibility, 41	CVodeGetDky, 62
CVLS linear solver interface	CVodeGetErrWeights, 69
convergence test, 59	CVodeGetEstLocalErrors, 69
Jacobian approximation used by, 55	CVodeGetIntegratorStats, 70
Jacobian-vector product approximation used	CVodeGetLastLinFlag, 76
by, 57	CVodeGetLastOrder, 67
memory requirements, 73	CVodeGetLastStep, 67
optional input, 54–60	CVodeGetLinReturnFlagName, 77
optional output, 73–77	CVodeGetLinSolvStats, 76
preconditioner setup function, 58, 87	CVodeGetLinWorkSpace, 73
preconditioner solve function, 58, 87	CVodeGetNonlinearSystemData, 3, 345
CVLS_ILL_INPUT, 41, 55, 56, 58, 59, 90, 95	CVodeGetNonlinSolvStats, 71
CVLS_JACFUNC_RECVR, 83, 85	CVodeGetNumErrTestFails, 67
CVLS_JACFUNC_UNRECVR, 83, 85	CVodeGetNumGEvals, 72
CVLS_LMEM_NULL, 55–59, 73–77, 90, 95	CVodeGetNumJacEvals, 73
CVLS_MEM_FAIL, 42, 90, 95	CVodeGetNumJtimesEvals, 76
CVLS_MEM_NULL, 41, 55–59, 73–77, 90, 95, 96	
	CVodeGetNumJTSetupEvals, 75
CVLS_PMEM_NULL, 90, 91, 95, 96	CVodeGetNumLinConvFails, 74
CVLS_SUCCESS, 41, 55–59, 73–77, 90, 91, 95, 96	CVodeGetNumLinIters, 74
CVLS_SUNLS_FAIL, 42, 57, 59	CVodeGetNumLinRhsEvals, 74
CVLsJacFn, 83	CVodeGetNumLinSolvSetups, 66
CVLsJacTimesSetupFn, 86	CVodeGetNumNonlinSolvConvFails, 70
CVLsJacTimesVecFn, 85	CVodeGetNumNonlinSolvIters, 70
CVLsLinSysFn, 85	CVodeGetNumPrecEvals, 75
CVLsPrecSetupFn, 87	CVodeGetNumPrecSolves, 75
CVLsPrecSolveFn, 87	CVodeGetNumProjEvals, 72
CVMonitorFn, 81	CVodeGetNumProjFails, 72
CVODE, 1	CVodeGetNumRhsEvals, 66
motivation for writing in C, 2	CVodeGetNumStabLimOrderReds, 68
package structure, 28	CVodeGetNumSteps, 66
relationship to CVODE, PVODE, 1–2	CVodeGetReturnFlagName, 71
relationship to VODE, VODPK, 1	CVodeGetRootInfo, 71
CVODE linear solver interfaces, 29	CVodeGetTolScaleFactor, 69
CVDIAG, 42	CVodeGetWorkSpace, 65
CVLS, 41	${\tt CVodeInit}, 38, 53, 79$
selecting one, 41	CVodeReInit, 79
CVODE linear solvers	${\tt CVodeRootInit},43$
header files, 33	${ t CVodeSetConstraints}, { t 53}$
implementation details, 29–30	${ t CVodeSetEpsLin}, { t 59}$
NVECTOR compatibility, 31	CVodeSetEpsProj, 62
selecting one, 41	CVodeSetErrFile, 46
$CVode,\ 36,\ 44$	CVodeSetErrHandlerFn, 46
cvode/cvode.h, 33	${\tt CVodeSetInitStep}, 50$
cvode/cvode_diag.h, 34	${ t CVodeSetJacEvalFrequency,} 55$
cvode/cvode_ls.h, 33	CVodeSetJacFn, 55
CVodeComputeState, 3	CVodeSetJacTimes, 57
CVodeCreate, 38	CVodeSetJacTimesRhsFn, 58
CVodeFree, 36, 38	CVodeSetLinearSolutionScaling, 56
CVodeGetActualInitStep, 68	CVodeSetLinearSolver, 36, 41, 83, 225
CVodeGetCurrentGamma, 345	CVodeSetLinSysFn, 56
CVodeGetCurrentOrder, 67	CVodeSetLSetupFrequency, 54
CVodeGetCurrentState, 345	CVodeSetLSNormFactor, 59
CVodeGetCurrentStep, 68	CVodeSetMaxConvFails, 52
CVodeGetCurrentTime 68	CVodeSetMaxErrTestFails 52

CVodeSetMaxHnilWarns, 50	step size selection, 22
CVodeSetMaxNonlinIters, 52	error messages, 46
CVodeSetMaxNumProjFails, 61	redirecting, 46
CVodeSetMaxNumSteps, 49	user-defined handler, 46, 80
CVodeSetMaxOrd, 49	
CVodeSetMaxStep, 51	FCVBANDSETJAC, 109
CVodeSetMinStep, 51	FCVBBDINIT, 119
CVodeSetMonitorFn, 48	FCVBBDOPT, 119
CVodeSetMonitorFrequency, 49	FCVBBDREINIT, 119
CVodeSetNoInactiveRootWarn, 60	FCVBJAC, 109
CVodeSetNonlinConvCoef, 52	FCVBPINIT, 117
CVodeSetNonLinearSolver, 43	FCVBPOPT, 117
CVodeSetNonlinearSolver, 36, 42	FCVCOMMFN, 120
CVodeSetPreconditioner, 58	FCVDENSESETJAC, 109
CVodeSetProjErrEst, 61	FCVDIAG, 112
CVodeSetProjFailEta, 62	FCVDJAC, 108
CVodeSetProjFn, 44	FCVDKY, 113
CVodeSetProjFrequency, 61	FCVDLSINIT, 108
CVodeSetRootDirection, 60	FCVEWT, 108
CVodeSetStabLimDet, 50	FCVEWTSET, 108
CVodeSetStopTime, 51	FCVFREE, 113
CVodeSetUseIntegratorFusedKernels, 53	FCVFUN, 106
CVodeSetUserData, 48	FCVGETERRWEIGHTS, 114
CVodeSStolerances, 39	FCVGETESTLOCALERR, 116
CVodeSVtolerances, 39	FCVGLOCFN, 120
CVodeWFtolerances, 39	FCVJTIMES, 110 , 120
CVProjFn, 82	FCVJTSETUP, 110 , 120
CVRhsFn, 38, 58, 80	FCVLSINIT, 108
CVRootFn, 82	FCVLSSETJAC, 111, 117, 119
CVSpilsGetLastFlag, 77	FCVLSSETPREC, 112
CVSpilsGetNumConvFails, 75	FCVMALLOC, 108
CVSpilsGetNumJtimesEvals, 76	FCVMALLOC, 107
CVSpilsGetNumJTSetupEvals, 76	FCVODE, 113
CVSpilsGetNumLinIters, 74	FCVODE interface module
CVSpilsGetNumPrecEvals, 75	interface to the CVBANDPRE module, 117–118
CVSpilsGetNumPrecSolves, 75	interface to the CVBBDPRE module, 118–120
CVSpilsGetNumRhsEvals, 74	optional input and output, 114
CVSpilsGetReturnFlagName, 77	rootfinding, 116
CVSpilsGetWorkspace, 73	usage, $105-113$
CVSpilsJacTimesSetupFn, 87	user-callable functions, 104–105
CVSpilsJacTimesVecFn, 86	user-supplied functions, 105
CVSpilsPrecSetupFn, 88	fcvode_mod, 97
CVSpilsPrecSolveFn, 87	FCVPSET, 111
CVSpilsSetEpsLin, 59	FCVPSOL, 111
CVSpilsSetJacTimes, 57	FCVREINIT, 113
CVSpilsSetLinearSolver, 42	FCVSETIIN, 114
CVSpilsSetPreconditioner, 59	FCVSETRIN, 114
-	FCVSETVIN, 114
data types	FCVSPARSESETJAC, 110
Fortran, 103	FCVSPILSETJAC, 111
	FCVSPILSETPREC, 112
eh_data, 81	FCVSPILSINIT, 108
error control	FCVSPJAC, 110
order selection, 22–23	fnvector_serial_mod, 151

FSUNBANDLINSOLINIT, 273	FSUNSPFGMRINIT, 309
FSUNDENSELINSOLINIT, 271	FSUNSPFGMRSETGSTYPE, 309
FSUNFIXEDPOINTINIT, 353	FSUNSPFGMRSETMAXRS, 310
FSUNKLUINIT, 283	FSUNSPFGMRSETPRECTYPE, 310
FSUNKLUREINIT, 283	FSUNSPGMRINIT, 301
FSUNKLUSETORDERING, 284	FSUNSPGMRSETGSTYPE, 302
FSUNLAPACKBANDINIT, 278	FSUNSPGMRSETMAXRS, 303
FSUNLAPACKDENSEINIT, 276	FSUNSPGMRSETPRECTYPE, 302
fsunlinsol_band_mod, 273	FSUNSPTFQMRINIT, 322
fsunlinsol_dense_mod, 271	FSUNSPTFQMRSETMAXL, 323
fsunlinsol_klu_mod, 283	FSUNSPTFQMRSETPRECTYPE, 323
fsunlinsol_pcg_mod, 329	FSUNSUPERLUMTINIT, 291
fsunlinsol_spbcgs_mod, 316	FSUNSUPERLUMTSETORDERING, 292
fsunlinsol_spfgmr_mod, 308	,
fsunlinsol_spgmr_mod, 301	half-bandwidths, 90, 94
fsunlinsol_sptfqmr_mod, 322	header files, 33, 89, 93
FSUNMASSBANDLINSOLINIT, 274	HNIL_WARNS, 114
FSUNMASSDENSELINSOLINIT, 271	
FSUNMASSKLUINIT, 283	INIT_STEP, 114
FSUNMASSKLUREINIT, 284	IOUT, 114, 115
FSUNMASSKLUSETORDERING, 284	itask, 36, 44
FSUNMASSLAPACKBANDINIT, 278	
FSUNMASSLAPACKDENSEINIT, 276	Jacobian approximation function
FSUNMASSPCGINIT, 329	band
FSUNMASSPCGSETMAXL, 330	use in FCVODE, 109
FSUNMASSPCGSETPRECTYPE, 330	dense
	use in fcvode, 108
FSUNMASSSPBCGSINIT, 316	diagonal
FSUNMASSSPBCGSSETMAXL, 317	difference quotient, 42
FSUNMASSSPBCGSSETPRECTYPE, 317	difference quotient, 55
FSUNMASSSPFGMRINIT, 309	Jacobian times vector
FSUNMASSSPFGMRSETGSTYPE, 309	alternative-rhs, 57
FSUNMASSSPFGMRSETMAXRS, 310	difference quotient, 57
FSUNMASSSPFGMRSETPRECTYPE, 310	use in fcvode, 110
FSUNMASSSPGMRINIT, 301	user-supplied, 57
FSUNMASSSPGMRSETGSTYPE, 302	Jacobian-vector product
FSUNMASSSPGMRSETMAXRS, 303	user-supplied, 85–86
FSUNMASSSPGMRSETPRECTYPE, 303	Jacobian-vector setup, 86–87
FSUNMASSSPTFQMRINIT, 322	sparse
FSUNMASSSPTFQMRSETMAXL, 324	use in fcvode, 110
FSUNMASSSPTFQMRSETPRECTYPE, 323	user-supplied, 55 , $83-84$
FSUNMASSSUPERLUMTINIT, 292	Jacobian update frequency
FSUNMASSUPERLUMTSETORDERING, 292	optional input, 54
fsunmatrix_band_mod, 235	Jacobian-vector product
fsunmatrix_dense_mod, 229	setup and solve phases, 29
fsunmatrix_sparse_mod, 242	1 ,
FSUNNEWTONINIT, 349	Linear solution scaling function
fsunnonlinsol_fixedpoint_mod, 353	user-supplied, 56
$fsunnonlinsol_newton_mod, 349$	Linear solver setup frequency
FSUNPCGINIT, 329	optional input, 54
FSUNPCGSETMAXL, 330	Linear system approximation function
FSUNPCGSETPRECTYPE, 330	user-supplied, 56
FSUNSPBCGSINIT, 316	Linear system function
FSUNSPBCGSSETMAXL, 317	user-supplied, 84–85
FSUNSPBCGSSETPRECTYPE, 317	1mm, 38, 79

LSODE, 1	N_VEnableConstVectorArray_Cuda, 178
	N_VEnableConstVectorArray_Hip, 184
MAX_CONVFAIL, 114	N_VEnableConstVectorArray_ManyVector, 207
MAX_ERRFAIL, 114	N_VEnableConstVectorArray_MPIManyVector, 212
MAX_NITERS, 114	N_VEnableConstVectorArray_OpenMP, 160
MAX_NSTEPS, 114	N_VEnableConstVectorArray_OpenMPDEV, 201
MAX_ORD, 114	N_VEnableConstVectorArray_Parallel, 155
MAX_STEP, 114	N_VEnableConstVectorArray_ParHyp, 170
maxord, 49, 79	N_VEnableConstVectorArray_Petsc, 173
memory requirements	N_VEnableConstVectorArray_Pthreads, 166
CVBANDPRE preconditioner, 91	N_VEnableConstVectorArray_Raja, 190
CVBBDPRE preconditioner, 96	N_VEnableConstVectorArray_Serial, 149
CVDIAG linear solver interface, 77	N_VEnableConstVectorArray_Sycl, 195
CVLS linear solver interface, 73	N_VEnableDotProdMulti_Cuda, 178
CVODE solver, 65	
MIN_STEP, 114	N_VEnableDotProdMulti_Hip, 184
monitor, 81	N_VEnableDotProdMulti_ManyVector, 206
,	N_VEnableDotProdMulti_MPIManyVector, 211
N_VCloneVectorArray, 138	N_VEnableDotProdMulti_OpenMP, 160
N_VCloneVectorArray_OpenMP, 158	N_VEnableDotProdMulti_OpenMPDEV, 201
N_VCloneVectorArray_OpenMPDEV, 199	N_VEnableDotProdMulti_Parallel, 155
N_VCloneVectorArray_Parallel, 153	N_VEnableDotProdMulti_ParHyp, 169
N_VCloneVectorArray_ParHyp, 168	N_VEnableDotProdMulti_Petsc, 173
N_VCloneVectorArray_Petsc, 172	N_VEnableDotProdMulti_Pthreads, 165
N_VCloneVectorArray_Pthreads, 164	N_VEnableDotProdMulti_Serial, 149
N_VCloneVectorArray_Serial, 147	N_VEnableFusedOps_Cuda, 178
N_VCloneVectorArrayEmpty, 138	N_VEnableFusedOps_Hip, 184
N_VCloneVectorArrayEmpty_OpenMP, 158	N_VEnableFusedOps_ManyVector, 206
N_VCloneVectorArrayEmpty_OpenMPDEV, 199	N_VEnableFusedOps_MPIManyVector, 211
N_VCloneVectorArrayEmpty_Parallel, 153	$ t N_VEnableFusedOps_OpenMP, 159$
N_VCloneVectorArrayEmpty_ParHyp, 168	$ t N_VEnableFusedOps_OpenMPDEV, 200$
N_VCloneVectorArrayEmpty_Petsc, 172	N_VEnableFusedOps_Parallel, 154
N_VCloneVectorArrayEmpty_Pthreads, 164	N_VEnableFusedOps_ParHyp, 169
N_VCloneVectorArrayEmpty_Serial, 148	N_VEnableFusedOps_Petsc, 172
N_VCopyFromDevice_Cuda, 177	N_VEnableFusedOps_Pthreads, 164
N_VCopyFromDevice_Hip, 183	N_VEnableFusedOps_Raja, 190
N_VCopyFromDevice_OpenMPDEV, 200	N_VEnableFusedOps_Serial, 148
N_VCopyFromDevice_Raja, 189	N_VEnableFusedOps_Sycl, 194
N_VCopyFromDevice_Sycl, 193	N_VEnableLinearCombination_Cuda, 178
N_VCopyOps, 138	N_VEnableLinearCombination_Hip, 184
N_VCopyToDevice_Cuda, 177	N_VEnableLinearCombination_ManyVector, 206
N_VCopyToDevice_Hip, 183	N_VEnableLinearCombination_MPIManyVector, 211
N_VCopyToDevice_OpenMPDEV, 200	N_VEnableLinearCombination_OpenMP, 159
N_VCopyToDevice_Raja, 189	N_VEnableLinearCombination_OpenMPDEV, 200
N_VCopyToDevice_Syc1, 193	N_VEnableLinearCombination_Parallel, 154
N_VDestroyVectorArray, 138	N_VEnableLinearCombination_ParHyp, 169
N_VDestroyVectorArray_OpenMP, 159	N_VEnableLinearCombination_Petsc, 172
N_VDestroyVectorArray_OpenMPDEV, 199	N_VEnableLinearCombination_Pthreads, 165
N_VDestroyVectorArray_Parallel, 153	N_VEnableLinearCombination_Raja, 190
N_VDestroyVectorArray_ParHyp, 168	N_VEnableLinearCombination_Serial, 149
N_VDestroyVectorArray_Petsc, 172	N_VEnableLinearCombination_Sycl, 195
N_VDestroyVectorArray_Pthreads, 164	N_VEnableLinearCombinationVectorArray_Cuda,
N_VDestroyVectorArray_Serial, 148	179
N_Vector, 33, 123, 140	N_VEnableLinearCombinationVectorArray_Hip, 185
14_400001, 30, 120, 140	w_vanabrearnearoombrhacronveccorarray_nrp, 100

```
{\tt N\_VEnableLinearCombinationVectorArray\_OpenMPN\_VEnableScaleAddMultiVectorArray\_Pthreads},
       161
                                                    166
N_VEnableLinearCombinationVectorArray_OpenMRDEVEnableScaleAddMultiVectorArray_Raja, 191
                                            N_VEnableScaleAddMultiVectorArray_Serial, 150
N_VEnableLinearCombinationVectorArray_Paral NeVEnableScaleAddMultiVectorArray_Sycl, 195
                                            N_VEnableScaleVectorArray_Cuda, 178
N_VEnableLinearCombinationVectorArray_ParHypN_VEnableScaleVectorArray_Hip, 184
                                            N_VEnableScaleVectorArray_ManyVector, 207
N_VEnableLinearCombinationVectorArray_Petsc, N_VEnableScaleVectorArray_MPIManyVector, 212
                                            N_VEnableScaleVectorArray_OpenMP, 160
N_VEnableLinearCombinationVectorArray_PthreaddyEnableScaleVectorArray_OpenMPDEV, 201
                                            N_VEnableScaleVectorArray_Parallel, 155
N_VEnableLinearCombinationVectorArray_Raja, N_VEnableScaleVectorArray_ParHyp, 169
                                            N_VEnableScaleVectorArray_Petsc, 173
N_VEnableLinearCombinationVectorArray_SerialN_VEnableScaleVectorArray_Pthreads, 165
                                            N_VEnableScaleVectorArray_Raja, 190
N_VEnableLinearCombinationVectorArray_Sycl, N_VEnableScaleVectorArray_Serial, 149
       195
                                            N_VEnableScaleVectorArray_Sycl, 195
N_VEnableLinearSumVectorArray_Cuda, 178
                                            N_VEnableWrmsNormMaskVectorArray_Cuda, 179
N_VEnableLinearSumVectorArray_Hip, 184
                                            N_VEnableWrmsNormMaskVectorArray_Hip, 185
N_VEnableLinearSumVectorArray_MPIManyVector,
                                                   207
       212
                                            N_VEnableWrmsNormMaskVectorArray_MPIManyVector,
N_VEnableLinearSumVectorArray_OpenMP, 160
N_VEnableLinearSumVectorArray_OpenMPDEV, 201 N_VEnableWrmsNormMaskVectorArray_OpenMP, 161
N_VEnableLinearSumVectorArray_Parallel, 155 N_VEnableWrmsNormMaskVectorArray_OpenMPDEV,
N_VEnableLinearSumVectorArray_ParHyp, 169
N_VEnableLinearSumVectorArray_Petsc, 173
                                            N_VEnableWrmsNormMaskVectorArray_Parallel, 155
N_VEnableLinearSumVectorArray_Pthreads, 165
                                            N_VEnableWrmsNormMaskVectorArray_ParHyp, 170
N_VEnableLinearSumVectorArray_Raja, 190
                                            N_VEnableWrmsNormMaskVectorArray_Petsc, 173
N_VEnableLinearSumVectorArray_Serial, 149
                                            N_VEnableWrmsNormMaskVectorArray_Pthreads, 166
N_VEnableLinearSumVectorArray_Sycl, 195
                                            N_VEnableWrmsNormMaskVectorArray_Serial, 150
N_VEnableScaleAddMulti_Cuda, 178
                                            N_VEnableWrmsNormVectorArray_Cuda, 179
N_VEnableScaleAddMulti_Hip, 184
                                            N_VEnableWrmsNormVectorArray_Hip, 185
N_VEnableScaleAddMulti_ManyVector, 206
                                            N_VEnableWrmsNormVectorArray_ManyVector, 207
N_VEnableScaleAddMulti_MPIManyVector, 211
                                            N_VEnableWrmsNormVectorArray_MPIManyVector,
N_VEnableScaleAddMulti_OpenMP, 160
N_VEnableScaleAddMulti_OpenMPDEV, 200
                                            N_VEnableWrmsNormVectorArray_OpenMP, 160
N_VEnableScaleAddMulti_Parallel, 154
                                            N_VEnableWrmsNormVectorArray_OpenMPDEV, 201
N_VEnableScaleAddMulti_ParHyp, 169
                                            N_VEnableWrmsNormVectorArray_Parallel, 155
                                            N_VEnableWrmsNormVectorArray_ParHyp, 170
N_VEnableScaleAddMulti_Petsc, 173
N_VEnableScaleAddMulti_Pthreads, 165
                                            N_VEnableWrmsNormVectorArray_Petsc, 173
N_VEnableScaleAddMulti_Raja, 190
                                            N_VEnableWrmsNormVectorArray_Pthreads, 166
                                            N_VEnableWrmsNormVectorArray_Serial, 150
N_VEnableScaleAddMulti_Serial, 149
N_VEnableScaleAddMulti_Sycl, 195
                                            N_VGetArrayPointer_MPIPlusX, 214
                                            N_VGetDeviceArrayPointer_Cuda, 175
N_VEnableScaleAddMultiVectorArray_Cuda, 179
N_VEnableScaleAddMultiVectorArray_Hip, 185
                                            N_VGetDeviceArrayPointer_Hip, 182
N_VEnableScaleAddMultiVectorArray_OpenMP, 161N_VGetDeviceArrayPointer_OpenMPDEV, 199
N_VEnableScaleAddMultiVectorArray_OpenMPDEV, N_VGetDeviceArrayPointer_Raja, 188
                                            N_VGetDeviceArrayPointer_Sycl, 193
N_VEnableScaleAddMultiVectorArray_Parallel, N_VGetHostArrayPointer_Cuda, 175
                                            N_VGetHostArrayPointer_Hip, 182
N_VEnableScaleAddMultiVectorArray_ParHyp, 17(N_VGetHostArrayPointer_OpenMPDEV, 199
N_VEnableScaleAddMultiVectorArray_Petsc, 174 N_VGetHostArrayPointer_Raja, 188
```

N_VGetHostArrayPointer_Sycl, 193	N_VNewEmpty_Petsc, 171
N_VGetLocalLength_Parallel, 153	N_VNewEmpty_Pthreads, 163
N_VGetLocalVector_MPIPlusX, 214	N_VNewEmpty_Raja, 189
N_VGetNumSubvectors_ManyVector, 205	N_VNewEmpty_SensWrapper, 342
N_VGetNumSubvectors_MPIManyVector, 211	N_VNewEmpty_Serial, 147
N_VGetSubvector_ManyVector, 205	N_VNewEmpty_Syc1, 193
N_VGetSubvector_MPIManyVector, 210	N_VNewManaged_Cuda, 176
N_VGetSubvectorArrayPointer_ManyVector, 205	
N_VGetSubvectorArrayPointer_MPIManyVector, 2	
N_VGetVector_ParHyp, 168	N_VNewManaged_Syc1, 192
N_VGetVector_Petsc, 171	N_VNewWithMemHelp_Cuda, 176
N_VGetVector_Trilinos, 203	N_VNewWithMemHelp_Raja, 188
N_VIsManagedMemory_Cuda, 175	N_VNewWithMemHelp_Syc1, 193
N_VIsManagedMemory_Hip, 182	N_VPrint_Cuda, 177
N_VIsManagedMemory_Raja, 188	N_VPrint_Hip, 183
N_VIsManagedMemory_Syc1, 194	N_VPrint_OpenMP, 159
N_VMake_Cuda, 176	N_VPrint_OpenMPDEV, 200
N_VMake_Hip, 182	N_VPrint_Parallel, 154
N_VMake_MPIManyVector, 210	N_VPrint_ParHyp, 168
N_VMake_MPIPlusX, 213	N_VPrint_Petsc, 172
N_VMake_OpenMP, 158	
N_VMake_OpenMPDEV, 199	N_VPrint_Pthreads, 164
- · · · · · · · · · · · · · · · · · · ·	N_VPrint_Raja, 189 N_VPrint_Serial, 148
N_VMake_Parallel, 153 N_VMake_ParHyp, 168	.4
01	N_VPrint_Sycl, 194 N_VPrintFile_Cude_177
N_VMake_Petsc, 171	N_VPrintFile_Cuda, 177
N_VMake_Pthreads, 163	N_VPrintFile_Hip, 183
N_VMake_Raja, 189	N_VPrintFile_OpenMP, 159
N_VMake_Serial, 147	N_VPrintFile_OpenMPDEV, 200
N_VMake_Syc1, 192	N_VPrintFile_Parallel, 154
N_VMake_Trilinos, 203	N_VPrintFile_ParHyp, 169
N_VMakeManaged_Cuda, 176	N_VPrintFile_Petsc, 172
N_VMakeManaged_Hip, 183	N_VPrintFile_Pthreads, 164
N_VMakeManaged_Raja, 189	N_VPrintFile_Raja, 189
N_VMakeManaged_Syc1, 192	N_VPrintFile_Serial, 148
N_VMakeWithManagedAllocator_Cuda, 176	N_VPrintFile_Sycl, 194
N_VNew_Cuda, 176	N_VSetArrayPointer_MPIPlusX, 214
N_VNew_Hip, 182	N_VSetCudaStream_Cuda, 177
N_VNew_ManyVector, 204	N_VSetDeviceArrayPointer_Cuda, 175
N_VNew_MPIManyVector, 209, 210	N_VSetDeviceArrayPointer_Raja, 188
N_VNew_OpenMP, 158	N_VSetDeviceArrayPointer_Syc1, 193
N_VNew_OpenMPDEV, 199	N_VSetHostArrayPointer_Cuda, 175
N_VNew_Parallel, 152	N_VSetHostArrayPointer_Raja, 188
N_VNew_Pthreads, 163	N_VSetHostArrayPointer_Syc1, 193
N_VNew_Raja, 188	N_VSetKernelExecPolicy_Cuda, 177
N_VNew_SensWrapper, 343	N_VSetKernelExecPolicy_Hip, 183
N_VNew_Serial, 147	N_VSetKernelExecPolicy_Sycl, 194
N_VNew_Sycl, 192	${\tt N_VSetSubvectorArrayPointer_ManyVector}, 205$
N_VNewEmpty, 138	N_VSetSubvectorArrayPointer_MPIManyVector, 210
N_VNewEmpty_Cuda, 176	NLCONV_COEF, 114
N_VNewEmpty_Hip, 182	nonlinear system
N_VNewEmpty_OpenMP, 158	Convergence test, 21
N_VNewEmpty_OpenMPDEV, 199	definition, 19–20
N_VNewEmpty_Parallel, 153	Newton iteration, 20–21
N_VNewEmpty_ParHyp, 168	${\tt NV_COMM_P},152$

NV_CONTENT_OMP, 157	Preconditioner setup routine
NV_CONTENT_OMPDEV, 198	use in fcvode, 111
NV_CONTENT_P, 151	Preconditioner solve routine
NV_CONTENT_PT, 162	use in fcvode, 111
NV_CONTENT_S, 146	Preconditioner update frequency
NV_DATA_DEV_OMPDEV, 198	optional input, 54
NV_DATA_HOST_OMPDEV, 198	preconditioning
NV_DATA_OMP, 157	advice on, 24, 29
NV_DATA_P, 152	band-block diagonal, 91
NV_DATA_PT, 162	banded, 89
NV_DATA_S, 146	setup and solve phases, 29
NV_GLOBLENGTH_P, 152	user-supplied, $58-59$, 87
NV_Ith_OMP, 157	Projection, 43
NV_Ith_P, 152	PVODE, 1
NV_Ith_PT, 163	
NV_Ith_S, 147	RCONST, 32
NV_LENGTH_OMP, 157	realtype, 32
NV_LENGTH_OMPDEV, 198	reinitialization, 79
NV_LENGTH_PT, 162	right-hand side function, 80
NV_LENGTH_S, 146	Rootfinding, 26, 36, 43, 116
NV_LOCLENGTH_P, 152	ROUT, 114, 115
NV_NUM_THREADS_OMP, 157	, , ,
NV_NUM_THREADS_PT, 162	SM_COLS_B, 232
NV_OWN_DATA_OMP, 157	SM_COLS_D, 227
NV_OWN_DATA_OMPDEV, 198	SM_COLUMN_B, 84, 232
NV_OWN_DATA_P, 152	SM_COLUMN_D, 83, 227
NV_OWN_DATA_PT, 162	SM_COLUMN_ELEMENT_B, 84, 232
NV_OWN_DATA_S, 146	SM_COLUMNS_B, 232
NVECTOR module, 123	SM_COLUMNS_D, 227
nvector_openmp_mod, 161	SM_COLUMNS_S, 239
nvector_pthreads_mod, 167	SM_CONTENT_B, 230
nvoodor_ponroudo_mod; rov	SM_CONTENT_D, 227
optional input	SM_CONTENT_S, 237
generic linear solver interface, 54–60	SM_DATA_B, 232
iterative linear solver, 58–60	SM_DATA_D, 227
matrix-based linear solver, 55–57	SM_DATA_S, 239
matrix-free linear solver, 57–58	SM_ELEMENT_B, 84, 232
projection, 60–62	SM_ELEMENT_D, 83, 227
rootfinding, 60	SM_INDEXPTRS_S, 239
solver, 46–53	SM_INDEXVALS_S, 239
optional output	SM_LBAND_B, 232
band-block-diagonal preconditioner, 95–96	SM_LDATA_B, 232
banded preconditioner, 90–91	SM_LDATA_D, 227
diagonal linear solver interface, 77–79	SM_LDIM_B, 232
generic linear solver interface, 73–77	SM_NNZ_S, 84, 239
interpolated solution, 62	SM_NP_S, 239
projection, 72	SM_ROWS_B, 232
rootfinding, 71–72	SM_ROWS_D, 227
solver, 65–71	SM_ROWS_S, 239
version, 63–65	SM_SPARSETYPE_S, 239
output mode, 23, 44	SM_SUBAND_B, 232
output mode, 20, 44	SM_UBAND_B, 232
portability, 32	SMALL_REAL, 32
Fortran, 103	STAB_LIM, 114
I O I O I O I I I I I I I I I I I I I I	~

Stability limit detection, 25	SUNLinSol_KLUReInit, 281
step size bounds, 50–51	SUNLinSol_KLUSetOrdering, 283
STOP_TIME, 114	SUNLinSol_LapackBand, 41, 277
SUNBandMatrix, 35, 233	SUNLinSol_LapackDense, 41, 275
SUNBandMatrix_Cols, 234	SUNLinSol_MagmaDense, 296
SUNBandMatrix_Column, 235	SUNLinSol_PCG, 41, 326, 329, 330
SUNBandMatrix_Columns, 234	SUNLinSol_PCGSetMax1, 328
SUNBandMatrix_Data, 234	SUNLinSol_PCGSetPrecType, 327
SUNBandMatrix_LDim, 234	SUNLinSol_SPBCGS, 41, 313, 316
SUNBandMatrix_LowerBandwidth, 234	SUNLinSol_SPBCGSSetMax1, 314
SUNBandMatrix_Print, 233	SUNLinSol_SPBCGSSetPrecType, 314
SUNBandMatrix_Rows, 233	SUNLinSol_SPFGMR, 41, 305, 309
SUNBandMatrix_StoredUpperBandwidth, 234	SUNLinSol_SPFGMRSetMaxRestarts, 307
SUNBandMatrix_UpperBandwidth, 234	SUNLinSol_SPFGMRSetPrecType, 306, 307
SUNBandMatrixStorage, 233	SUNLinSol_SPGMR, 41, 298, 301, 302
SUNDenseMatrix, 35, 228	SUNLinSol_SPGMRSetMaxRestarts, 300
SUNDenseMatrix_Cols, 228	SUNLinSol_SPGMRSetPrecType, 299
SUNDenseMatrix_Column, 229	SUNLinSol_SPTFQMR, 41, 319, 322, 323
SUNDenseMatrix_Columns, 228	SUNLinSol_SPTFQMRSetMax1, 321
SUNDenseMatrix_Data, 228	SUNLinSol_SPTFQMRSetPrecType, 320
SUNDenseMatrix_LData, 228	SUNLinSol_SuperLUDIST, 287
SUNDenseMatrix_Print, 228	SUNLinSol_SuperLUDIST_GetBerr, 287
SUNDenseMatrix_Rows, 228	SUNLinSol_SuperLUDIST_GetGridinfo, 287
sundials/sundials_linearsolver.h, 255	SUNLinSol_SuperLUDIST_GetLUstruct, 288
sundials_nonlinearsolver.h, 33	SUNLinSol_SuperLUDIST_GetScalePermstruct, 288
sundials_nvector.h, 33	SUNLinSol_SuperLUDIST_GetSOLVEstruct, 288
sundials_types.h, 32, 33	SUNLinSol_SuperLUDIST_GetSuperLUOptions, 288
SUNDIALSGetVersion, 63	SUNLinSol_SuperLUDIST_GetSuperLUStat, 288
SUNDIALSGetVersionNumber, 65	SUNLinSol_SuperLUMT, 41, 290
sunindextype, 32	SUNLinSol_SuperLUMTSetOrdering, 292
SUNLinearSolver, 41, 255, 263	SUNLinSolFree, 37, 256, 258
SUNLinearSolver module, 255	SUNLinSolGetID, 256, 257
SUNLINEARSOLVER_DIRECT, 83, 257, 266	SUNLinSolGetType, 256, 257
	SUNLinSolInitialize, 256, 257
SUNLINEARSOLVER_ITERATIVE, 257, 267 SUNLINEARSOLVER_MATRIX_ITERATIVE, 257, 267	SUNLinSollastFlag, 260
sunlinsol/sunlinsol_band.h, 33	SUNLinSolNewEmpty, 265
	SUNLinSolNumIters, 260
sunlinsol/sunlinsol_dense.h, 33 sunlinsol/sunlinsol_klu.h, 33	,
	SUNLinSolResNorm, 260
sunlinsol/sunlinsol_lapackband.h, 33	SUNLinSolSetATimes, 257–259, 267
sunlinsol/sunlinsol_lapackdense.h, 33	SUNLinSolSetInfoFile_PCG, 328
sunlinsol/sunlinsol_pcg.h, 34	SUNLinSolSetInfoFile_SPBCGS, 315
sunlinsol/sunlinsol_spbcgs.h, 34	SUNLinSolSetInfoFile_SPFGMR, 307
sunlinsol/sunlinsol_spfgmr.h, 34	SUNLinSolSetInfoFile_SPGMR, 300
sunlinsol/sunlinsol_spgmr.h, 33	SUNLinSolSetInfoFile_SPTFQMR, 321
sunlinsol/sunlinsol_sptfqmr.h, 34	SUNLinSolSetPreconditioner, 259
sunlinsol/sunlinsol_superlumt.h, 33	SUNLinSolSetPrintLevel_PCG, 328
SUNLinSol_Band, 41, 272	SUNLinSolSetPrintLevel_SPBCGS, 315
SUNLinSol_cuSolverSp_batchQR, 294	SUNLinSolSetPrintLevel_SPFGMR, 308
SUNLinSol_cuSolverSp_batchQR_GetDescription	
295	SUNLinSolSetPrintLevel_SPTFQMR, 321
SUNLinSol_cuSolverSp_batchQR_SetDescription	·
295	SUNLinSolSetup, 256, 258, 267
SUNLinSol Dense, 41, 270	SUNLinSolSolve, 256, 258
SUNLinSol_KLU, 41, 280	SUNLinSolSpace, 261

SUNMatCopyOps, 222	${\tt SUNSparseMatrix_NP}, {\tt 241}$
SUNMatDestroy, 37	${\tt SUNSparseMatrix_Print}, {\tt 241}$
SUNMatNewEmpty, 222	${\tt SUNSparseMatrix_Realloc}, 240$
SUNMatrix, 219, 224	SUNSparseMatrix_Reallocate, 240
SUNMatrix module, 219	${\tt SUNSparseMatrix_Rows},241$
SUNMatrix_SLUNRloc, 243	SUNSparseMatrix_SparseType, 241
SUNMatrix_SLUNRloc_OwnData, 244	
SUNMatrix_SLUNRloc_Print, 244	tolerances, 20, 40, 81
SUNMatrix_SLUNRloc_ProcessGrid, 244	
SUNMatrix_SLUNRloc_SuperMatrix, 244	UNIT_ROUNDOFF, 32
SUNMemory, 359	User main program
SUNMemory module, 359	CVBANDPRE usage, 89
SUNMemoryHelper, 359	CVBBDPRE usage, 93
SUNMemoryType, 359	FCVBBD usage, 118
SUNNonlinearSolver, 33, 333	FCVBP usage, 117
SUNNonlinearSolver module, 333	FCVODE usage, 105
SUNNONLINEARSOLVER_FIXEDPOINT, 334	IVP solution, 34
SUNNONLINEARSOLVER_ROOTFIND, 334	$user_data, 48, 80-83, 92, 93$
SUNNonlinSol_FixedPoint, 352, 355	
SUNNonlinSol_FixedPointSens, 352	VODE, 1
SUNNonlinSol_Newton, 347	VODPK, 1
SUNNonlinSol_NewtonSens, 347	waighted root mean gauge norm 20
SUNNonlinSol_PetscSNES, 356	weighted root-mean-square norm, 20
SUNNonlinSolFree, 37, 335	
SUNNonlinSolGetCurIter, 337	
SUNNonlinSolGetNumConvFails, 338	
SUNNonlinSolGetNumIters, 337	
SUNNonlinSolGetPetscError_PetscSNES, 357	
SUNNonlinSolGetSNES_PetscSNES, 357	
SUNNonlinSolGetSysFn_FixedPoint, 352	
SUNNonlinSolGetSysFn_Newton, 348	
SUNNonlinSolGetSysFn_PetscSNES, 357	
SUNNonlinSolGetType, 334	
SUNNonlinSolInitialize, 334	
SUNNonlinSollSetupFn, 336	
SUNNonlinSolNewEmpty, 344	
SUNNonlinSolSetConvTestFn, 336	
SUNNonlinSolSetInfoFile FixedPoint, 353	
SUNNonlinSolSetInfoFile Newton, 348	
SUNNonlinSolSetLSolveFn, 336	
SUNNonlinSolSetMaxIters, 337	
SUNNonlinSolSetPrintLevel_FixedPoint, 354	
SUNNonlinSolSetPrintLevel_Newton, 349	
SUNNonlinSolSetSysFn, 335	
SUNNonlinSolSetup, 334	
SUNNonlinSolSolve, 334	
SUNSparseFromBandMatrix, 240	
SUNSparseFromDenseMatrix, 240	
SUNSparseMatrix, 35, 239	
SUNSparseMatrix_Columns, 241	
SUNSparseMatrix_Data, 241	
SUNSparseMatrix_IndexPointers, 242	
SUNSparseMatrix_IndexValues, 242	
SUNSparseMatrix_NNZ, 84, 241	