

User Documentation for `CVODE` v4.0.0-dev.2 (SUNDIALS v4.0.0-dev.2)

Alan C. Hindmarsh and Radu Serban
*Center for Applied Scientific Computing
Lawrence Livermore National Laboratory*

Daniel R. Reynolds
*Department of Mathematics
Southern Methodist University*

September 28, 2018



UCRL-SM-208108

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.

Contents

List of Tables	vii
List of Figures	ix
1 Introduction	1
1.1 Historical Background	1
1.2 Changes from previous versions	2
1.3 Reading this User Guide	9
1.4 SUNDIALS Release License	10
1.4.1 Copyright Notices	10
1.4.1.1 SUNDIALS Copyright	10
1.4.1.2 ARKode Copyright	11
1.4.2 BSD License	11
2 Mathematical Considerations	13
2.1 IVP solution	13
2.2 Preconditioning	17
2.3 BDF stability limit detection	18
2.4 Rootfinding	19
3 Code Organization	21
3.1 SUNDIALS organization	21
3.2 CVODE organization	21
4 Using CVODE for C Applications	25
4.1 Access to library and header files	25
4.2 Data Types	26
4.2.1 Floating point types	26
4.2.2 Integer types used for vector and matrix indices	26
4.3 Header files	27
4.4 A skeleton of the user's main program	28
4.5 User-callable functions	31
4.5.1 CVODE initialization and deallocation functions	32
4.5.2 CVODE tolerance specification functions	33
4.5.3 Linear solver interface functions	34
4.5.4 Nonlinear solver interface function	36
4.5.5 Rootfinding initialization function	37
4.5.6 CVODE solver function	37
4.5.7 Optional input functions	39
4.5.7.1 Main solver optional input functions	39
4.5.7.2 Linear solver interface optional input functions	45
4.5.7.3 Rootfinding optional input functions	48
4.5.8 Interpolated output function	49

4.5.9	Optional output functions	50
4.5.9.1	SUNDIALS version information	50
4.5.9.2	Main solver optional output functions	52
4.5.9.3	Rootfinding optional output functions	58
4.5.9.4	CVLS linear solver interface optional output functions	59
4.5.9.5	Diagonal linear solver interface optional output functions	63
4.5.10	CVODE reinitialization function	64
4.6	User-supplied functions	65
4.6.1	ODE right-hand side	65
4.6.2	Error message handler function	66
4.6.3	Error weight function	66
4.6.4	Rootfinding function	67
4.6.5	Jacobian construction (matrix-based linear solvers)	67
4.6.6	Jacobian-vector product (matrix-free linear solvers)	69
4.6.7	Jacobian-vector product setup (matrix-free linear solvers)	69
4.6.8	Preconditioner solve (iterative linear solvers)	70
4.6.9	Preconditioner setup (iterative linear solvers)	71
4.7	Preconditioner modules	72
4.7.1	A serial banded preconditioner module	72
4.7.2	A parallel band-block-diagonal preconditioner module	74
5	FCVODE, an Interface Module for FORTRAN Applications	81
5.1	Important note on portability	81
5.2	Fortran Data Types	81
5.3	FCVODE routines	82
5.4	Usage of the FCVODE interface module	83
5.5	FCVODE optional input and output	92
5.6	Usage of the FCVROOT interface to rootfinding	94
5.7	Usage of the FCVBP interface to CVBANDPRE	95
5.8	Usage of the FCVBBD interface to CVBBDPRE	96
6	Description of the NVECTOR module	99
6.1	The NVECTOR_SERIAL implementation	108
6.2	The NVECTOR_PARALLEL implementation	110
6.3	The NVECTOR_OPENMP implementation	113
6.4	The NVECTOR_PTHREADS implementation	116
6.5	The NVECTOR_PARHYP implementation	118
6.6	The NVECTOR_PETSC implementation	120
6.7	The NVECTOR_CUDA implementation	121
6.8	The NVECTOR_RAJA implementation	124
6.9	NVECTOR Examples	127
6.10	NVECTOR functions used by CVODE	130
7	Description of the SUNMatrix module	133
7.1	The SUNMatrix_Dense implementation	136
7.2	The SUNMatrix_Band implementation	139
7.3	The SUNMatrix_Sparse implementation	143
7.4	SUNMatrix Examples	149
7.5	SUNMatrix functions used by CVODE	150

8	Description of the SUNLinearSolver module	153
8.0.1	SUNLinearSolver core functions	154
8.0.2	SUNLinearSolver set functions	155
8.0.3	SUNLinearSolver get functions	156
8.0.4	Functions provided by SUNDIALS packages	158
8.0.5	SUNLinearSolver return codes	159
8.0.6	The generic SUNLinearSolver module	160
8.1	Compatibility of SUNLinearSolver modules	161
8.2	Implementing a custom SUNLinearSolver module	161
8.3	The SUNLinearSolver_Dense implementation	162
8.3.1	SUNLINSOL_DENSE usage	162
8.3.2	SUNLINSOL_DENSE description	163
8.4	The SUNLinearSolver_Band implementation	164
8.4.1	SUNLINSOL_BAND usage	164
8.4.2	SUNLINSOL_BAND description	165
8.5	The SUNLinearSolver_LapackDense implementation	166
8.5.1	SUNLINSOL_LAPACKDENSE usage	166
8.5.2	SUNLINSOL_LAPACKDENSE description	167
8.6	The SUNLinearSolver_LapackBand implementation	168
8.6.1	SUNLINSOL_LAPACKBAND usage	168
8.6.2	SUNLINSOL_LAPACKBAND description	169
8.7	The SUNLinearSolver_KLU implementation	170
8.7.1	SUNLINSOL_KLU usage	170
8.7.2	SUNLINSOL_KLU description	174
8.8	The SUNLinearSolver_SuperLUMT implementation	175
8.8.1	SUNLINSOL_SUPERLUMT usage	175
8.8.2	SUNLINSOL_SUPERLUMT description	178
8.9	The SUNLinearSolver_SPGMR implementation	179
8.9.1	SUNLINSOL_SPGMR usage	180
8.9.2	SUNLINSOL_SPGMR description	183
8.10	The SUNLinearSolver_SPFGMR implementation	185
8.10.1	SUNLINSOL_SPFGMR usage	186
8.10.2	SUNLINSOL_SPFGMR description	189
8.11	The SUNLinearSolver_SPBCGS implementation	191
8.11.1	SUNLINSOL_SPBCGS usage	192
8.11.2	SUNLINSOL_SPBCGS description	195
8.12	The SUNLinearSolver_SPTFQMR implementation	196
8.12.1	SUNLINSOL_SPTFQMR usage	196
8.12.2	SUNLINSOL_SPTFQMR description	199
8.13	The SUNLinearSolver_PCG implementation	201
8.13.1	SUNLINSOL_PCG usage	202
8.13.2	SUNLINSOL_PCG description	205
8.14	SUNLinearSolver Examples	206
8.15	SUNLinearSolver functions used by CVODE	207
9	Description of the SUNNonlinearSolver module	209
9.1	The SUNNonlinearSolver API	209
9.1.1	SUNNonlinearSolver core functions	209
9.1.2	SUNNonlinearSolver set functions	211
9.1.3	SUNNonlinearSolver get functions	212
9.1.4	Functions provided by SUNDIALS integrators	213
9.1.5	SUNNonlinearSolver return codes	215
9.1.6	The generic SUNNonlinearSolver module	215
9.1.7	Usage with sensitivity enabled integrators	216

9.1.8	Implementing a Custom SUNNonlinearSolver Module	217
9.2	The SUNNonlinearSolver_Newton implementation	218
9.2.1	SUNNonlinearSolver_Newton description	218
9.2.2	SUNNonlinearSolver_Newton functions	219
9.2.3	SUNNonlinearSolver_Newton content	220
9.2.4	SUNNonlinearSolver_Newton Fortran interface	220
9.3	The SUNNonlinearSolver_FixedPoint implementation	220
9.3.1	SUNNonlinearSolver_FixedPoint description	221
9.3.2	SUNNonlinearSolver_FixedPoint functions	221
9.3.3	SUNNonlinearSolver_FixedPoint content	223
9.3.4	SUNNonlinearSolver_FixedPoint Fortran interface	224
A	SUNDIALS Package Installation Procedure	225
A.1	CMake-based installation	226
A.1.1	Configuring, building, and installing on Unix-like systems	226
A.1.2	Configuration options (Unix/Linux)	228
A.1.3	Configuration examples	234
A.1.4	Working with external Libraries	235
A.1.5	Testing the build and installation	237
A.2	Building and Running Examples	237
A.3	Configuring, building, and installing on Windows	238
A.4	Installed libraries and exported header files	238
B	CVODE Constants	243
B.1	CVODE input constants	243
B.2	CVODE output constants	243
	Bibliography	247
	Index	251

List of Tables

4.1	SUNDIALS linear solver interfaces and vector implementations that can be used for each.	31
4.2	Optional inputs for CVODE and CVLS	40
4.3	Optional outputs from CVODE, CVLS, and CVDIAG	51
5.1	Keys for setting FCVODE optional inputs	92
5.2	Description of the FCVODE optional output arrays IOUT and ROUT	93
6.1	Vector Identifications associated with vector kernels supplied with SUNDIALS.	101
6.2	Description of the NVECTOR operations	102
6.3	Description of the NVECTOR fused operations	105
6.4	Description of the NVECTOR vector array operations	106
6.5	List of vector functions usage by CVODE code modules	130
7.1	Identifiers associated with matrix kernels supplied with SUNDIALS.	134
7.2	Description of the SUNMatrix operations	134
7.3	SUNDIALS matrix interfaces and vector implementations that can be used for each. . .	135
7.4	List of matrix functions usage by CVODE code modules	150
8.1	Description of the SUNLinearSolver error codes	159
8.2	SUNDIALS matrix-based linear solvers and matrix implementations that can be used for each.	161
8.3	List of linear solver functions usage by CVODE code modules	208
9.1	Description of the SUNNonlinearSolver return codes	215
A.1	SUNDIALS libraries and header files	238

List of Figures

3.1	High-level diagram of the SUNDIALS suite	22
3.2	Organization of the SUNDIALS suite	23
3.3	Overall structure diagram of the CVODE package	24
7.1	Diagram of the storage for a SUNMATRIX_BAND object	140
7.2	Diagram of the storage for a compressed-sparse-column matrix	146
A.1	Initial <i>ccmake</i> configuration screen	227
A.2	Changing the <i>instdir</i>	228

Chapter 1

Introduction

CVODE is part of a software family called SUNDIALS: SUite of Nonlinear and Differential/ALgebraic equation Solvers [25]. 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 [5] and VODPK [8]. 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 [33]. 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 [6]. The capabilities of both VODE and VODPK have been combined in the C-language package CVODE [13].

At present, CVODE may utilize a variety of Krylov methods provided in SUNDIALS that can be used in conjunction with Newton iteration: these include the GMRES (Generalized Minimal RESidual) [36], FGMRES (Flexible Generalized Minimum RESidual) [35], Bi-CGStab (Bi-Conjugate Gradient Stabilized) [37], TFQMR (Transpose-Free Quasi-Minimal Residual) [19], and PCG (Preconditioned Conjugate Gradient) [20] 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 multiprocessor environments with minimal impacts on the rest of the solver, resulting in PVIDE [11], 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 Hypré 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 v4.0.0-dev.2

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. The user interface for the new CVLS module is very similar to the previous CVDLS and CVSPILS interfaces; however 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. We do 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 `SUNLinSol_Band`, `SUNLinSol_Dense`, `SUNLinSol_KLU`, `SUNLinSol_LapackBand`, `SUNLinSol_LapackDense`, `SUNLinSol_PCG`, `SUNLinSol_SPBCGS`, `SUNLinSol_SPGMR`, `SUNLinSol_SPGMR`, `SUNLinSol_SPTFQMR`, and `SUNLinSol_SuperLUMT`. Solver-specific “set” routine names have been similarly standardized. To minimize challenges in user migration to the new names, the previous routine names may still be used; these will be deprecated in future releases, so we recommend that users migrate to the new names soon.

Changes in v4.0.0-dev.1

An API for encapsulating the nonlinear solvers used in SUNDIALS implicit integrators has been introduced. The goal of this API is to ease the introduction of new nonlinear solver options in SUNDIALS integrators and allow for external or user-supplied nonlinear solvers. The SUNNONLINSOL API and provided SUNNONLINSOL modules are described in Chapter 9 and follow the same object oriented design and implementation used by the NVECTOR, SUNMATRIX, and SUNLINSOL modules.

SUNNONLINSOL modules are intended to solve nonlinear systems formulated as either a rootfinding problem $F(y) = 0$ or a fixed-point problem $G(y) = y$. 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. Additionally, the fixed-point iteration can use 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 as well as 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. To use the fixed-point nonlinear solver with CVODE a user must do the following:

1. include the header `sunnonlinsol/sunnonlinsol_fixedpoint.h` to access the `SUNNONLINSOL_FIXEDPOINT` module,
2. create a nonlinear solver object by calling the fixed-point constructor

```
NLS = SUNNonlinSol_FixedPoint(y, m);
```

where `NLS` is the nonlinear solver object of type `SUNNonlinearSolver`, `y` is an `NVector` the size of the ODE system, and `m` is the number of acceleration vectors (see §9.3 for more detail), and

3. after initializing CVODE with `CVodeInit`, call

```
retval = CVodeSetNonlinearSolver(cvode_mem, NLS);
```

to attach the nonlinear solver object to the CVODE memory structure.

User-supplied nonlinear solver modules adhering to the `SUNNONLINSOL` API for solving $F(y) = 0$ or $G(y) = y$ can be used with CVODE by following the same include, create, and attach process just described for the `SUNNONLINSOL_FIXEDPOINT` module.

As a result of this new nonlinear solver object creation and attachment approach, the function `CVodeSetIterType` has been removed. 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.

Changes in v4.0.0-dev

Three fused vector operations and seven vector array operations have been added to the `NVECTOR` API. These *optional* 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 new operations are `N_VLinearCombination`, `N_VScaleAddMulti`, `N_VDotProdMulti`, `N_VLinearCombinationVectorArray`, `N_VScaleVectorArray`, `N_VConstVectorArray`, `N_VWrmsNormVectorArray`, `N_VWrmsNormMaskVectorArray`, `N_VScaleAddMultiVectorArray`, and `N_VLinearCombinationVectorArray`. If any of these operations are defined as `NULL` in an `NVECTOR` implementation the `NVECTOR` interface will automatically call standard `NVECTOR` operations as necessary. Details on the new operations can be found in Chapter 6.

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 §4.5.7.1 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 deprecated. The new options that align with those used in native CMake FindMPI module are `MPI_C_COMPILER`, `MPI_CXX_COMPILER`, `MPI_Fortran_COMPILER`, and `MPIEXEC_EXECUTABLE`.
- When a Fortran name-mangling scheme is needed (e.g., `LAPACK_ENABLE` 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 SUNLINEARSOLVER module to use a `typedef` for the precision-specific solve function to be used (to avoid compiler warnings).
- Added missing typecasts for some `(void*)` pointers (again, to avoid compiler warnings).
- Bugfix in `sunmatrix_sparse.c` where we had used `int` instead of `sunindextype` in one location.
- Added missing `#include <stdio.h>` in 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 object-oriented 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 object-oriented API.
- Added example problems demonstrating use of generic SUNLINEARSOLVER modules.
- Expanded package-provided direct linear solver (Dls) interfaces and scaled, preconditioned, iterative linear solver (Spils) interfaces to utilize generic SUNMATRIX and SUNLINEARSOLVER objects.

- Removed package-specific, linear solver-specific, solver modules (e.g. CVDENSE, KINBAND, IDAKLU, ARKSPGMR) since their functionality is entirely replicated by the generic Dls/Spils interfaces and SUNLINEARSOLVER/SUNMATRIX modules. The exception is CVDIAG, a diagonal approximate Jacobian solver available to CVODE and CVODES.
- Converted all SUNDIALS example problems to utilize new generic SUNMATRIX and SUNLINEARSOLVER objects, along with updated Dls and Spils linear solver interfaces.
- Added Spils interface routines to ARKode, CVODE, CVODES, IDA and IDAS to allow specification of a user-provided "JTSetup" routine. This change supports users who wish to set up data structures for the user-provided Jacobian-times-vector ("JTimes") routine, and where the cost of one JTSetup setup per Newton iteration can be amortized between multiple JTimes calls.

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

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

To avoid potential namespace conflicts, the macros defining `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 `lfree` 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 `init` 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`, `SUNRpowerI`, 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 installation *include* directory.

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

Changes in v2.4.0

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 SUNDIALS suite. We have eliminated the mechanism of providing optional inputs and extracting optional statistics from the solver through the `iopt` and `ropt` arrays. Instead, 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 §4.5.7 and §4.5.9.

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 §4.5.6 only, and looking at examples in [27].

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 (§4.5.10), (d) supply a new NVECTOR module (Chapter 6), (e) supply new SUNLINSOL and/or SUNMATRIX modules (Chapters 7 and 8), or even (f) supply new SUNNONLINSOL modules (Chapter 9).

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 (§2.2), stability limit detection (§2.3), and rootfinding (§2.4).
- 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 FCVODE, an interface module for the use of CVODE with FORTRAN applications.
- Chapter 6 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 7 gives a brief overview of the generic SUNMATRIX module shared among the various components of SUNDIALS, and details on the SUNMATRIX implementations provided with SUNDIALS: a dense implementation (§7.1), a banded implementation (§7.2) and a sparse implementation (§7.3).
- Chapter 8 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 9 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 [12] and George D. Byrne [10].

1.4 SUNDIALS Release License

The SUNDIALS packages are released open source, under a BSD license. The only requirements of the BSD license are preservation of copyright and a standard disclaimer of liability. Our Copyright notice is below along with the license.



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

1.4.1 Copyright Notices

All SUNDIALS packages except ARKode are subject to the following Copyright notice.

1.4.1.1 SUNDIALS Copyright

Copyright (c) 2002-2016, Lawrence Livermore National Security. Produced at the Lawrence Livermore National Laboratory. Written by A.C. Hindmarsh, D.R. Reynolds, R. Serban, C.S. Woodward, S.D. Cohen, A.G. Taylor, S. Peles, L.E. Banks, and D. Shumaker.

UCRL-CODE-155951 (CVODE)

UCRL-CODE-155950 (CVODES)

UCRL-CODE-155952 (IDA)

UCRL-CODE-237203 (IDAS)

LLNL-CODE-665877 (KINSOL)
All rights reserved.

1.4.1.2 ARKode Copyright

ARKode is subject to the following joint Copyright notice. Copyright (c) 2015-2016, Southern Methodist University and Lawrence Livermore National Security Written by D.R. Reynolds, D.J. Gardner, A.C. Hindmarsh, C.S. Woodward, and J.M. Sexton.
LLNL-CODE-667205 (ARKODE)
All rights reserved.

1.4.2 BSD License

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

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the disclaimer below.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the disclaimer (as noted below) in the documentation and/or other materials provided with the distribution.
3. Neither the name of the LLNS/LLNL 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 LAWRENCE LIVERMORE NATIONAL SECURITY, LLC, THE U.S. DEPARTMENT OF ENERGY 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.

Additional BSD Notice

1. This notice is required to be provided under our contract with the U.S. Department of Energy (DOE). This work was produced at Lawrence Livermore National Laboratory under Contract No. DE-AC52-07NA27344 with the DOE.
2. Neither the United States Government nor Lawrence Livermore National Security, LLC nor any of their employees, makes any warranty, express or implied, or assumes any 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.
3. Also, reference herein to any specific commercial products, process, or services 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.

Chapter 2

Mathematical Considerations

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

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

where $y \in \mathbf{R}^N$. 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.

2.1 IVP solution

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

$$\sum_{i=0}^{K_1} \alpha_{n,i} y^{n-i} + h_n \sum_{i=0}^{K_2} \beta_{n,i} \dot{y}^{n-i} = 0. \quad (2.2)$$

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 $K_2 = q$ 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 [9] and [29].

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

$$F(y^n) \equiv y^n - h_n \beta_{n,0} f(t_n, y^n) - a_n = 0, \quad (2.3)$$

or as a fixed-point problem

$$G(y^n) \equiv h_n \beta_{n,0} f(t_n, y^n) + a_n = y^n. \quad (2.4)$$

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 9). By default CVODE solves (2.3) with a *Newton iteration* which requires the solution of linear systems

$$M[y^{n(m+1)} - y^{n(m)}] = -F(y^{n(m)}), \quad (2.5)$$

in which

$$M \approx I - \gamma J, \quad J = \partial f / \partial y, \quad \text{and} \quad \gamma = h_n \beta_{n,0}. \quad (2.6)$$

The exact variation of the Newton iteration depends on the choice of linear solver and is discussed below and in §9.2. For nonstiff systems, a *fixed-point iteration* (previously referred to as a functional

iteration in this guide) solving (2.4) is also available. This involves evaluations of f only and can (optionally) use Anderson's method [3, 38, 18, 32] to accelerate convergence (see §9.3 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 (2.5) (e.g., the default Newton iteration), CVODE provides several linear solver choices, including the option of a user-supplied linear solver module (see Chapter 8). 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 [14, 1], or the thread-enabled SuperLU_MT sparse solver library [31, 16, 2] (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 [6].

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 $\|\cdot\|_{\text{WRMS}}$, for all error-like quantities. The multiplicative weights used are based on the current solution and on the relative and absolute tolerances input by the user, namely

$$W_i = 1/[\text{RTOL} \cdot |y_i| + \text{ATOL}_i]. \quad (2.7)$$

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

In the case of a matrix-based linear solver, the default Newton iteration is a Modified Newton iteration, in that the iteration matrix M is fixed throughout the nonlinear iterations. However, in 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 $\|y^{n(m)} - y^{n(0)}\| \leq \epsilon$. Letting y^n denote the exact solution of (2.3), 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_j 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 (2.7). In the dense case, this scheme requires N evaluations of f , one for each column of J . In the band case, the columns of J are computed in groups, by the Curtis-Powell-Reid algorithm, with the number of f evaluations equal to the bandwidth.

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

$$Jv = [f(t, y + \sigma v) - f(t, y)]/\sigma. \quad (2.8)$$

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

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

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

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

The local error test is simply $\|\text{LTE}\| \leq 1$. 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

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

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, $\text{LTE}(q')$, 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 [5, 8, 24]. They are also summarized in [25].

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: $y_i > 0$, $y_i < 0$, $y_i \geq 0$, or $y_i \leq 0$. The constraint satisfaction is tested after a successful nonlinear system solution. If any constraint fails, we declare a convergence failure of the 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).

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 Preconditioning

When using a nonlinear solver that requires the solution of the linear system (2.5) (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 A . In order to improve the convergence of the Krylov iteration, the preconditioner matrix P , or the product $P_L P_R$ in the last case, should in some sense approximate the system matrix A . Yet at the same time, in order to be cost-effective, the matrix P , 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 [6] 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 $P_L P_R$). 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.

2.3 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 [22]. 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 [23], 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.

2.4 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 [21]. 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

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

Whenever sign changes are seen in two or more root functions, the one deemed most likely to have its root occur first is the one with the largest value of $|g_i(t_{hi})|/|g_i(t_{hi}) - g_i(t_{lo})|$, corresponding to the closest to t_{lo} of the secant method values. At each pass through the loop, a new value t_{mid} is set, strictly within the search interval, and the values of $g_i(t_{mid})$ are checked. Then either t_{lo} or t_{hi} is reset to t_{mid} according to which subinterval is found to 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 Figs. 3.1 and 3.2). The 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 ODE systems $Mdy/dt = f_E(t, y) + f_I(t, y)$ based on additive 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 $F(u) = 0$.

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 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 8) and SUNNONLINSOL API (see Chapter 9) 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 8). These solvers may utilize a SUNMATRIX object (see Chapter 7) for storing Jacobian

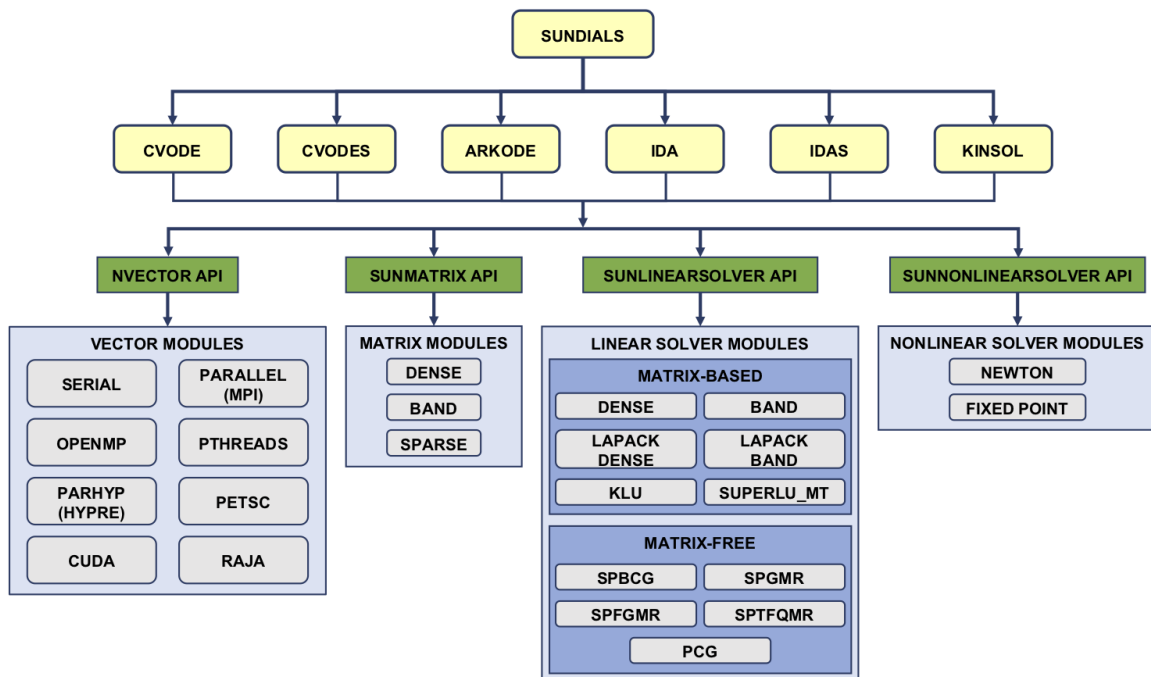


Figure 3.1: High-level diagram of the SUNDIALS suite

information, or they may be matrix-free. Since CVMODE can operate on any valid SUNLINSOL implementation, the set of linear solver modules available to CVMODE will expand as new SUNLINSOL modules are developed.

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

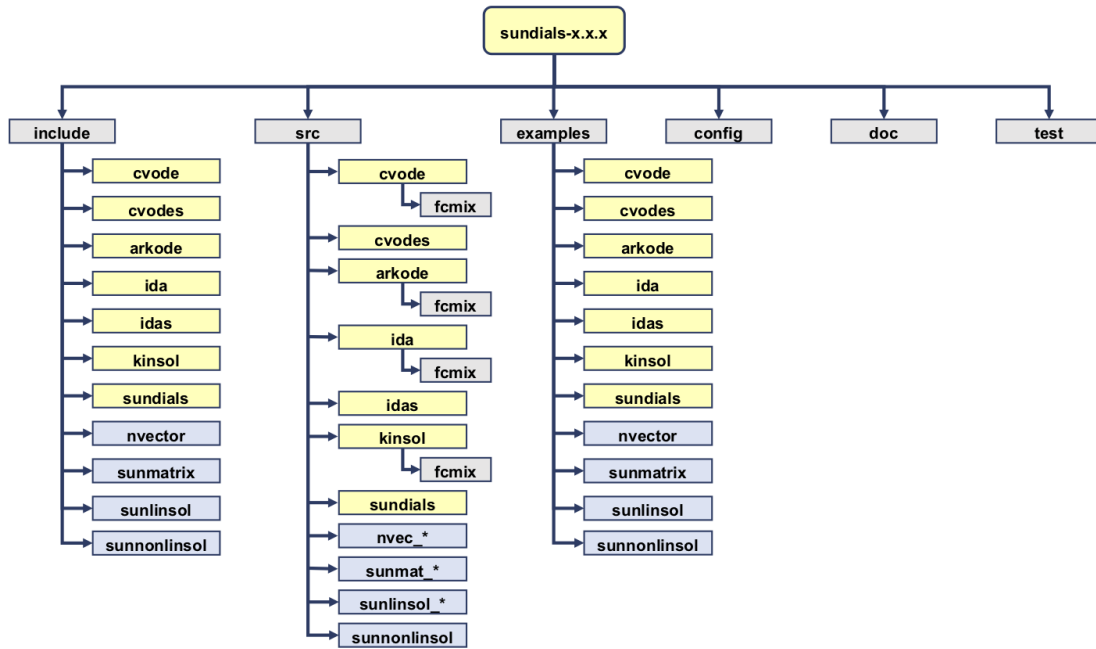
For users employing dense or banded Jacobian matrices, CVMODE includes algorithms for their approximation through difference quotients, although the user also has the option of supplying a routine to compute the Jacobian (or an approximation to it) directly. This user-supplied routine is required when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

For users employing matrix-free iterative linear solvers, CVMODE 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.

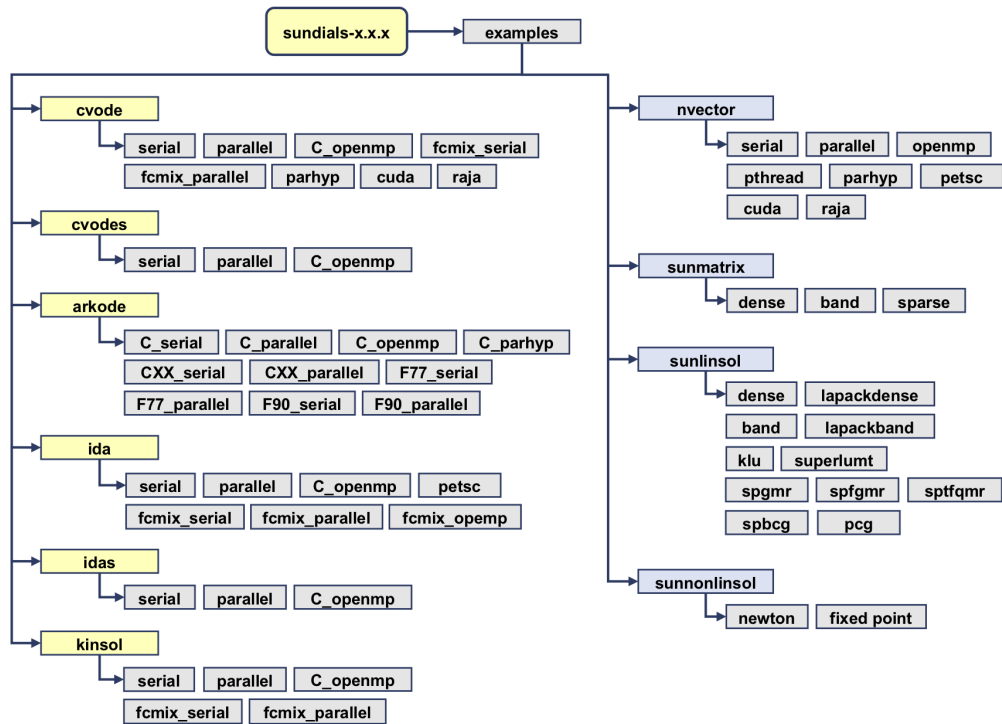
For preconditioned iterative methods, the preconditioning must be supplied by the user, again in two phases: setup and solve. While there is no default choice of preconditioner analogous to the difference-quotient approximation in the direct case, the references [6, 8], together with the example and demonstration programs included with CVMODE, offer considerable assistance in building preconditioners.

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

CVMODE 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



(a) Directory structure of the SUNDIALS source tree



(b) Directory structure of the SUNDIALS examples

Figure 3.2: Organization of the SUNDIALS suite

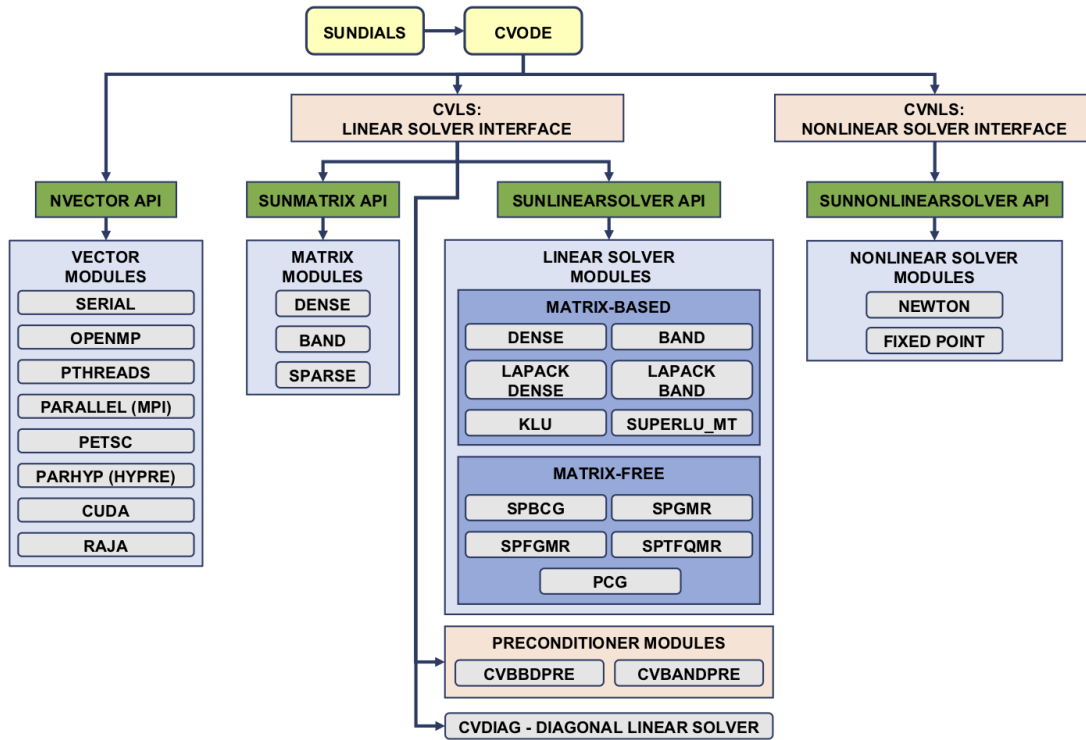


Figure 3.3: Overall structure diagram of the CVODE package. Modules specific to CVODE begin with “CV” (CVLS, CVDIAG, CVBBDPRE, CVBANDPRE, and CVNLS), all other items correspond to generic solver and auxiliary modules. Note also that the LAPACK, KLU and SUPERLUMT support is through interfaces to external packages. Users will need to download and compile those packages independently.

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 [27] 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 the FORTRAN/C interface module.

The user should be aware that not all SUNLINSOL and SUNMATRIX modules are compatible with all NVECTOR implementations. Details on compatibility are given in the documentation for each SUNMATRIX module (Chapter 7) and each SUNLINSOL module (Chapter 8). 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 7 and 8 to verify compatibility between these modules. In addition to that documentation, we note that the CVBANDPRE 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.

A user program which uses the type `realtype` and the `RCONST` macro to handle floating-point constants is precision-independent except for any calls to precision-specific standard math library functions. (Our example programs use both `realtype` and `RCONST`.) 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`). Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use `realtype`, so long as the SUNDIALS libraries use the correct precision (for details see §A.1.2).

4.2.2 Integer types used for vector and matrix indices

The type `sunindextype` can be either a 32- or 64-bit *signed* integer. The default is the portable `int64_t` type, and the user can change it to `int32_t` at the configuration stage. The configuration system will detect if the compiler does not support portable types, and will replace `int32_t` and `int64_t` with `int` and `long int`, respectively, to ensure use of the desired sizes on Linux, Mac OS X, and Windows platforms. SUNDIALS currently does not support *unsigned* integer types for vector and matrix indices, although these could be added in the future if there is sufficient demand.

A user program which uses `sunindextype` to handle vector and matrix 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 one of `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:

- `ccode/ccode.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, `ccode/ccode_ls.h`.

Note that `ccode.h` includes `sundials.types.h`, which defines the types `realtype`, `sunindextype`, and `boolean_type` 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 6 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/sunnonlinsol_***.h` where `***` is the name of the nonlinear solver module (see Chapter 9 for more information). This file in turn includes the header file `sundials_nonlinear_solver.h` which defines the abstract `SUNNonlinearSolver` data type.

If using a nonlinear solver that requires the solution of a linear system of the form (2.5) (e.g., the default Newton iteration), then a linear solver module header file will be required. The header files corresponding to the 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, `SUNLINSOL_DENSE`;
 - `sunlinsol/sunlinsol_band.h`, which is used with the banded linear solver module, `SUNLINSOL_BAND`;
 - `sunlinsol/sunlinsol_lapackdense.h`, which is used with the LAPACK dense linear solver interface module, `SUNLINSOL_LAPACKDENSE`;
 - `sunlinsol/sunlinsol_lapackband.h`, which is used with the LAPACK banded linear solver interface module, `SUNLINSOL_LAPACKBAND`;
 - `sunlinsol/sunlinsol_klu.h`, which is used with the KLU sparse linear solver interface module, `SUNLINSOL_KLU`;
 - `sunlinsol/sunlinsol_superlump.h`, which is used with the SUPERLUMP sparse linear solver interface module, `SUNLINSOL_SUPERLUMP`;
- 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_spgs.h`, which is used with the scaled, preconditioned Bi-CGStab Krylov linear solver module, `SUNLINSOL_SPGS`;

- `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 interface.

The header files for the `SUNLINSOL_DENSE` and `SUNLINSOL_LAPACKDENSE` linear solver modules include the file `sunmatrix/sunmatrix_dense.h`, which defines the `SUNMATRIX_DENSE` matrix module, as well as various functions and macros acting on such matrices.

The header files for the `SUNLINSOL_BAND` and `SUNLINSOL_LAPACKBAND` linear solver modules include the file `sunmatrix/sunmatrix_band.h`, which defines the `SUNMATRIX_BAND` matrix module, as well as various functions and macros acting on such matrices.

The header files for the `SUNLINSOL_KLU` and `SUNLINSOL_SUPERLUMT` sparse linear solvers include the file `sunmatrix/sunmatrix_sparse.h`, which defines the `SUNMATRIX_SPARSE` matrix module, as well as various functions and macros acting on such matrices.

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

Other headers may be needed, according to the choice of preconditioner, etc. For example, in the `cvDiurnal_kry_p` example (see [27]), 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 6, 7, 8, and 9 for the specific name of the function to be called or macro to be referenced.

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

For example, call `MPI_Init` to initialize MPI if used, 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 §6.1-6.4 for details.

For the *hypr* and PETSc vector wrappers, first create and initialize the underlying vector, and then create an `NVECTOR` wrapper with a call of the form `y0 = N_VMake_***(yvec)`, where `yvec` is a *hypr* or PETSc vector. Note that calls like `N_VNew_***(...)` and `N_VGetArrayPointer(...)` are not available for these vector wrappers. See §6.5 and §6.6 for details.

If using either the CUDA- or RAJA-based vector implementations use a call of the form `y0 = N_VMake_***(..., c)` where `c` is a pointer to a `suncudavec` or `sunrajavec` vector class if this class 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 where it is located with a call of the form `N_VGetDeviceArrayPointer_***` or `N_VGetHostArrayPointer_***`. Note that the vector class will allocate memory on both the host and device when instantiated. See §6.7-6.8 for details.

4. Create CVOICE object

Call `cvoice_mem = CVoiceCreate(lmm)` to create the CVOICE memory block and to specify the linear multistep method. `CVoiceCreate` returns a pointer to the CVOICE memory structure. See §4.5.1 for details.

5. Initialize CVOICE solver

Call `CVoiceInit(...)` to provide required problem specifications, allocate internal memory for CVOICE, and initialize CVOICE. `CVoiceInit` 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 `CVoiceSStolerances(...)` or `CVoiceSVtolerances(...)` 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 `CVoiceWftolerances` 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 §4.5.3 and Chapter 8.

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 8 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 §4.5.7.1 and §4.5.7 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 9 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 9 for more information on optional inputs.

15. Specify rootfinding problem

Optionally, call **CVodeRootInit** to initialize a rootfinding problem to be solved during the integration of the ODE system. See §4.5.5, and see §4.5.7.3 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 §4.5.6 for details.

17. Get optional outputs

Call **CV*Get*** functions to obtain optional output. See §4.5.9 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 8 the SUNDIALS packages operate on generic `SUNLINSOL` objects, allowing a user to develop their own solvers should they so desire.

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

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

4.5 User-callable functions

This section describes the `CVODE` functions that are called by the user to setup and then solve an IVP. Some of these are required. However, starting with §4.5.7, 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 §4.5.7.1).

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

Call	<code>cvode_mem = CVodeCreate(lmm);</code>
Description	The function <code>CVodeCreate</code> instantiates a CVODE solver object and specifies the solution method.
Arguments	<code>lmm</code> (<code>int</code>) specifies the linear multistep method and must be one of two possible values: <code>CV_ADAMS</code> or <code>CV_BDF</code> . The recommended choices for <code>lmm</code> are <code>CV_ADAMS</code> for nonstiff problems and <code>CV_BDF</code> 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 <code>CvodeSetNonlinearSolver</code> .
Return value	If successful, <code>CVodeCreate</code> returns a pointer to the newly created CVODE memory block (of type <code>void *</code>). Otherwise, it returns <code>NULL</code> .

CVodeInit

Call	<code>flag = CVodeInit(cvode_mem, f, t0, y0);</code>
Description	The function <code>CVodeInit</code> provides required problem and solution specifications, allocates internal memory, and initializes CVODE.
Arguments	<code>cvode_mem</code> (<code>void *</code>) pointer to the CVODE memory block returned by <code>CVodeCreate</code> . <code>f</code> (<code>CVRhsFn</code>) is the C function which computes the right-hand side function f in the ODE. This function has the form <code>f(t, y, ydot, user_data)</code> (for full details see §4.6.1). <code>t0</code> (<code>realtype</code>) is the initial value of t . <code>y0</code> (<code>N_Vector</code>) is the initial value of y .
Return value	The return value <code>flag</code> (of type <code>int</code>) will be one of the following: <code>CV_SUCCESS</code> The call to <code>CVodeInit</code> was successful. <code>CV_MEM_NULL</code> The CVODE memory block was not initialized through a previous call to <code>CVodeCreate</code> . <code>CV_MEM_FAIL</code> A memory allocation request has failed. <code>CV_ILL_INPUT</code> An input argument to <code>CVodeInit</code> has an illegal value.
Notes	If an error occurred, <code>CVodeInit</code> also sends an error message to the error handler function.

CVodeFree

Call	<code>CVodeFree(&cvode_mem);</code>
Description	The function <code>CVodeFree</code> frees the memory allocated by a previous call to <code>CVodeCreate</code> .
Arguments	The argument is the pointer to the CVODE memory block (of type <code>void *</code>).
Return value	The function <code>CVodeFree</code> has no return value.

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.

CVodeSVtolerances

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

Description The function `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.
- `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 .

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. (2.7).

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 §4.6.3).

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

- `CV_SUCCESS` The call to `CVodeWFtolerances` 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.

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 $1.0\text{E-}15$).

(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 [27]. In that problem, the three components vary between 0 and 1, and have different noise levels; hence the `abstol` vector. It is impossible to give any general advice on `abstol` values, because the appropriate noise levels are completely problem-dependent. The user or modeler hopefully has some idea as to what those noise levels are.

(3) Finally, it is important to pick all the tolerance values conservatively, because they control the error committed on each individual time step. The final (global) errors are 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 `y` 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 `y` vector) for the purposes of computing $f(t, y)$.

(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 (2.5) (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 SUNLINSOL 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 Jacobian on a vector, Jv . 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 SPFGMR that supports right preconditioning only and PCG that performs symmetric preconditioning. For the specification of a preconditioner, see the iterative linear solver sections in §4.5.7 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 $P_1 P_2$ approximates the matrix $M = I - \gamma J$ of (2.6).

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_SuperLUMT`, `SUNLinSol_SPGMR`, `SUNLinSol_SPFGMR`, `SUNLinSol_SPBCGS`, `SUNLinSol_SPTFQMR`, and `SUNLinSol_PCG`.

Alternately, a user-supplied `SUNLinearSolver` module may be created and used instead. The use of each of the generic linear solvers involves certain constants, functions and possibly some macros, that are likely to be needed in the user code. These are available in the corresponding header file associated with the specific SUNMATRIX or SUNLINSOL module in question, as described in Chapters 7 and 8.

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	<code>flag = CVodeSetLinearSolver(cvode_mem, LS, J);</code>
Description	The function <code>CVodeSetLinearSolver</code> attaches a generic SUNLINSOL object <code>LS</code> and corresponding template Jacobian SUNMATRIX object <code>J</code> (if applicable) to CVODE, initializing the CVLS linear solver interface.
Arguments	<p><code>cvode_mem</code> (void *) pointer to the CVODE memory block.</p> <p><code>LS</code> (<code>SUNLinearSolver</code>) SUNLINSOL object to use for solving linear systems of the form (2.5).</p> <p><code>J</code> (<code>SUNMatrix</code>) SUNMATRIX object for used as a template for the Jacobian (or <code>NULL</code> if not applicable).</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>CVLS_SUCCESS</code> The CVLS initialization was successful.</p>

	CVLS_MEM_NULL The <code>cvode_mem</code> 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	<p>If LS is a matrix-based linear solver, then the template Jacobian matrix J will be used in the solve process, so if additional storage is required within the SUNMATRIX object (e.g. for factorization of a banded matrix), ensure that the input object is allocated with sufficient size (see the documentation of the particular SUNMATRIX type in Chapter 7 for further information).</p> <p>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 CVOICE.</p> <p>The previous routines <code>CVdlsSetLinearSolver</code> and <code>CVSpilsSetLinearSolver</code> are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to the new routine name soon.</p>

CVDiag

Call	<code>flag = CVDiag(cvode_mem);</code>
Description	<p>The function <code>CVDiag</code> selects the CVDIAG linear solver.</p> <p>The user's main program must include the <code>cvode_diag.h</code> header file.</p>
Arguments	<code>cvode_mem</code> (<code>void *</code>) pointer to the CVOICE memory block.
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of:</p> <p>CVDIAG_SUCCESS The CVDIAG initialization was successful.</p> <p>CVDIAG_MEM_NULL The <code>cvode_mem</code> pointer is NULL.</p> <p>CVDIAG_ILL_INPUT The CVDIAG solver is not compatible with the current NVECTOR module.</p> <p>CVDIAG_MEM_FAIL A memory allocation request failed.</p>
Notes	The CVDIAG solver is the simplest of all of the available CVOICE linear solvers. The CVDIAG solver uses an approximate diagonal Jacobian formed by way of a difference quotient. The user does <i>not</i> have the option of supplying a function to compute an approximate diagonal Jacobian.

4.5.4 Nonlinear solver interface function

By default CVOICE uses the SUNNONLINSOL implementation of Newton's method defined by the `SUNNONLINSOL_NEWTON` module (see §9.2). To specify a different nonlinear solver in CVOICE, 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 CVOICE, `CVodeSetNonlinearSolver` must be called after `CVodeInit`. If any calls to `CVode` have been made, then CVOICE 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 CVOICE memory pointer returned by `CVodeCreate` and the second argument is the `SUNNONLINSOL` object to use for solving

the nonlinear system (2.3) or (2.4). 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 (2.3) or (2.4).

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.

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 §4.6.4 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`.

4.5.6 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	<code>flag = CVode(cvode_mem, tout, yout, &tret, itask);</code>
Description	The function <code>CVode</code> integrates the ODE over an interval in t .
Arguments	<p><code>cvode_mem</code> (void *) pointer to the CVODE memory block.</p> <p><code>tout</code> (realtype) the next time at which a computed solution is desired.</p> <p><code>yout</code> (N_Vector) the computed solution vector.</p> <p><code>tret</code> (realtype) the time reached by the solver (output).</p> <p><code>itask</code> (int) a flag indicating the job of the solver for the next user step. The <code>CV_NORMAL</code> option causes the solver to take internal steps until it has reached or just passed the user-specified <code>tout</code> parameter. The solver then interpolates in order to return an approximate value of $y(\text{tout})$. The <code>CV_ONE_STEP</code> option tells the solver to take just one internal step and then return the solution at the point reached by that step.</p>
Return value	<p><code>CVode</code> returns a vector <code>yout</code> and a corresponding independent variable value $t = \text{tret}$, such that <code>yout</code> is the computed value of $y(t)$.</p> <p>In <code>CV_NORMAL</code> mode (with no errors), <code>tret</code> will be equal to <code>tout</code> and <code>yout = y(tout)</code>.</p> <p>The return value <code>flag</code> (of type <code>int</code>) will be one of the following:</p> <p><code>CV_SUCCESS</code> <code>CVode</code> succeeded and no roots were found.</p> <p><code>CV_TSTOP_RETURN</code> <code>CVode</code> succeeded by reaching the stopping point specified through the optional input function <code>CVodeSetStopTime</code> (see §4.5.7.1).</p> <p><code>CV_ROOT_RETURN</code> <code>CVode</code> succeeded and found one or more roots. In this case, <code>tret</code> is the location of the root. If <code>nrtfn > 1</code>, call <code>CVodeGetRootInfo</code> to see which g_i were found to have a root.</p> <p><code>CV_MEM_NULL</code> The <code>cvode_mem</code> argument was <code>NULL</code>.</p> <p><code>CV_NO_MALLOC</code> The CVODE memory was not allocated by a call to <code>CVodeInit</code>.</p> <p><code>CV_ILL_INPUT</code> One of the inputs to <code>CVode</code> 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 <code>CVodeCreate</code>) failed to set the linear solver-specific <code>lsolve</code> field in <code>cvode_mem</code>. (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.</p> <p><code>CV_TOO_CLOSE</code> The initial time t_0 and the final time t_{out} are too close to each other and the user did not specify an initial step size.</p> <p><code>CV_TOO_MUCH_WORK</code> The solver took <code>mxstep</code> internal steps but still could not reach <code>tout</code>. The default value for <code>mxstep</code> is <code>MXSTEP_DEFAULT = 500</code>.</p> <p><code>CV_TOO_MUCH_ACC</code> The solver could not satisfy the accuracy demanded by the user for some internal step.</p> <p><code>CV_ERR_FAILURE</code> Either error test failures occurred too many times (<code>MXNEF = 7</code>) during one internal time step, or with $h = h_{min}$.</p> <p><code>CV_CONV_FAILURE</code> Either convergence test failures occurred too many times (<code>MXNCF = 10</code>) during one internal time step, or with $h = h_{min}$.</p> <p><code>CV_LINIT_FAIL</code> The linear solver interface's initialization function failed.</p> <p><code>CV_LSETUP_FAIL</code> The linear solver interface's setup function failed in an unrecoverable manner.</p> <p><code>CV_LSOLVE_FAIL</code> The linear solver interface's solve function failed in an unrecoverable manner.</p>

	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.
Notes		The vector yout can occupy the same space as the vector y0 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.
		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.

4.5.7 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, beginning 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. We also note that all error return values are negative, so the test **flag < 0** will catch all errors.

4.5.7.1 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.

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
CVLS linear solver interface		
Jacobian / preconditioner update frequency	CVodeSetMaxStepsBetweenJac	50
Jacobian function	CVodeSetJacFn	DQ
Jacobian-times-vector functions	CVodeSetJacTimes	NULL, DQ
Preconditioner functions	CVodeSetPreconditioner	NULL, NULL
Ratio between linear and nonlinear tolerances	CVodeSetEpsLin	0.05

CV_MEM_NULL The `ccode_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 CVOID memory pointer is NULL). This use of `CCodeSetErrFile` is strongly discouraged.

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



CCodeSetErrHandlerFn

Call `flag = CCodeSetErrHandlerFn(ccode_mem, ehfun, eh_data);`

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

Arguments `ccode_mem` (void *) pointer to the CVOID memory block.
`ehfun` (CErrorHandlerFn) 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 `ehfun` and data pointer `eh_data` have been successfully set.
CV_MEM_NULL The `ccode_mem` pointer is NULL.

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

CCodeSetUserData

Call `flag = CCodeSetUserData(ccode_mem, user_data);`

Description The function `CCodeSetUserData` specifies the user data block `user_data` and attaches it to the main CVOID memory block.

Arguments `ccode_mem` (void *) pointer to the CVOID 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 `ccode_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 `CCodeSetUserData` must be made *before* the call to specify the linear solver.



CCodeSetMaxOrd

Call `flag = CCodeSetMaxOrd(ccode_mem, maxord);`

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

Arguments `ccode_mem` (void *) pointer to the CVOID 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 `ccode_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.

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*).

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.

CVodeSetStabLimDet

Call `flag = CVodeSetstabLimDet(cvode_mem, stldet);`

Description The function `CVodeSetStabLimDet` indicates if the BDF stability limit detection algorithm should be used. See §2.3 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.

CVodeSetInitStep

Call	<code>flag = CVodeSetInitStep(cvode_mem, hin);</code>
Description	The function <code>CVodeSetInitStep</code> specifies the initial step size.
Arguments	<code>cvode_mem</code> (void *) pointer to the CVODE memory block. <code>hin</code> (realtype) value of the initial step size to be attempted. Pass 0.0 to use the default value.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of <code>CV_SUCCESS</code> The optional value has been successfully set. <code>CV_MEM_NULL</code> The <code>cvode_mem</code> pointer is <code>NULL</code> .
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 t_0 .

CVodeSetMinStep

Call	<code>flag = CVodeSetMinStep(cvode_mem, hmin);</code>
Description	The function <code>CVodeSetMinStep</code> specifies a lower bound on the magnitude of the step size.
Arguments	<code>cvode_mem</code> (void *) pointer to the CVODE memory block. <code>hmin</code> (realtype) minimum absolute value of the step size (≥ 0.0).
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of <code>CV_SUCCESS</code> The optional value has been successfully set. <code>CV_MEM_NULL</code> The <code>cvode_mem</code> pointer is <code>NULL</code> . <code>CV_ILL_INPUT</code> Either <code>hmin</code> is nonpositive or it exceeds the maximum allowable step size.
Notes	The default value is 0.0.

CVodeSetMaxStep

Call	<code>flag = CVodeSetMaxStep(cvode_mem, hmax);</code>
Description	The function <code>CVodeSetMaxStep</code> specifies an upper bound on the magnitude of the step size.
Arguments	<code>cvode_mem</code> (void *) pointer to the CVODE memory block. <code>hmax</code> (realtype) maximum absolute value of the step size (≥ 0.0).
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of <code>CV_SUCCESS</code> The optional value has been successfully set. <code>CV_MEM_NULL</code> The <code>cvode_mem</code> pointer is <code>NULL</code> . <code>CV_ILL_INPUT</code> Either <code>hmax</code> is nonpositive or it is smaller than the minimum allowable step size.
Notes	Pass <code>hmax = 0.0</code> to obtain the default value ∞ .

CVodeSetStopTime

Call	<code>flag = CVodeSetStopTime(cvode_mem, tstop);</code>
Description	The function <code>CVodeSetStopTime</code> specifies the value of the independent variable t past which the solution is not to proceed.
Arguments	<code>cvode_mem</code> (void *) pointer to the CVODE memory block. <code>tstop</code> (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.

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.

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.

CVodeSetMaxConvFails

Call `flag = CVodeSetMaxConvFails(cvode_mem, maxncf);`

Description The function `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.

CVodeSetNonlinConvCoef

Call	<code>flag = CVodeSetNonlinConvCoef(cvode_mem, nlscoef);</code>
Description	The function <code>CVodeSetNonlinConvCoef</code> specifies the safety factor used in the nonlinear convergence test (see §2.1).
Arguments	<code>cvode_mem</code> (<code>void *</code>) pointer to the CVODE memory block. <code>nlscoef</code> (<code>realtype</code>) coefficient in nonlinear convergence test (> 0.0).
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of <code>CV_SUCCESS</code> The optional value has been successfully set. <code>CV_MEM_NULL</code> The <code>cvode_mem</code> pointer is <code>NULL</code> .
Notes	The default value is 0.1.

CVodeSetConstraints

Call	<code>flag = CVodeSetConstraints(cvode_mem, constraints);</code>
Description	The function <code>CVodeSetConstraints</code> specifies a vector defining inequality constraints for each component of the solution vector y .
Arguments	<code>cvode_mem</code> (<code>void *</code>) pointer to the IDA memory block. <code>constraints</code> (<code>N_Vector</code>) vector of constraint flags. If <code>constraints[i]</code> 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 <code>flag</code> (of type <code>int</code>) is one of <code>CV_SUCCESS</code> The optional value has been successfully set. <code>CV_MEM_NULL</code> The <code>cvode_mem</code> pointer is <code>NULL</code> . <code>CV_ILL_INPUT</code> The constraints vector contains illegal values.
Notes	The presence of a non- <code>NULL</code> 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 <code>constraints</code> will result in an illegal input return. A <code>NULL</code> constraints vector will disable constraint checking.

4.5.7.2 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 a user-callable routine to modify this behavior. To this end, we recall that the Newton system matrices are $M(t, y) = I - \gamma J(t, y)$, where the right-hand side function has Jacobian matrix $J(t, y) = \frac{\partial f(t, y)}{\partial y}$.

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 `msbj` argument to `CVodeSetMaxStepsBetweenJac`.

CVodeSetMaxStepsBetweenJac

Call	<code>retval = CVodeSetMaxStepsBetweenJac(cvode_mem, msbj);</code>
Description	The function <code>CVodeSetMaxStepsBetweenJac</code> specifies the maximum number of time steps to wait before recomputation of the Jacobian or recommendation to update the preconditioner.
Arguments	<p><code>cvode_mem</code> (void *) pointer to the CVODE memory block.</p> <p><code>msbj</code> (long int) maximum number of time steps to wait before Jacobian/preconditioner reconstruction.</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>CVLS_SUCCESS</code> The optional value has been successfully set.</p> <p><code>CVLS_MEM_NULL</code> The <code>cvode_mem</code> pointer is <code>NULL</code>.</p> <p><code>CVLS_LMEM_NULL</code> The CVLS linear solver interface has not been initialized.</p>
Notes	<p>If <code>msbj</code> is less than 1, the default value of 50 will be used.</p> <p>This function must be called <i>after</i> the CVLS linear solver interface has been initialized through a call to <code>CVodeSetLinearSolver</code>.</p>

When using matrix-based linear solver modules, the CVLS solver interface needs a function to compute an approximation to the Jacobian matrix $J(t, y)$. This function 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	<code>flag = CVodeSetJacFn(cvode_mem, jac);</code>
Description	The function <code>CVodeSetJacFn</code> specifies the Jacobian approximation function to be used for a matrix-based solver within the CVLS interface.
Arguments	<p><code>cvode_mem</code> (void *) pointer to the CVODE memory block.</p> <p><code>jac</code> (<code>CVLSJacFn</code>) user-defined Jacobian approximation function.</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>CVLS_SUCCESS</code> The optional value has been successfully set.</p> <p><code>CVLS_MEM_NULL</code> The <code>cvode_mem</code> pointer is <code>NULL</code>.</p> <p><code>CVLS_LMEM_NULL</code> The CVLS linear solver interface has not been initialized.</p>
Notes	<p>This function must be called <i>after</i> the CVLS linear solver interface has been initialized through a call to <code>CVodeSetLinearSolver</code>.</p> <p>By default, CVLS uses an internal difference quotient function for dense and band matrices. If <code>NULL</code> is passed to <code>jac</code>, this default function is used. An error will occur if no <code>jac</code> is supplied when using other matrix types.</p> <p>The function type <code>CVLSJacFn</code> is described in §4.6.5.</p> <p>The previous routine <code>CVDlsSetJacFn</code> 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.</p>

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

The pointer `user_data` received through `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	<code>flag = CVodeSetJacTimes(cvode_mem, jtsetup, jtimes);</code>
Description	The function <code>CVSetJacTimes</code> specifies the Jacobian-vector setup and product functions.
Arguments	<code>cvode_mem</code> (void *) pointer to the CVODE memory block. <code>jtsetup</code> (<code>CVLSJacTimesSetupFn</code>) user-defined Jacobian-vector setup function. Pass <code>NULL</code> if no setup is necessary. <code>jtimes</code> (<code>CVLSJacTimesVecFn</code>) user-defined Jacobian-vector product function.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of <code>CVLS_SUCCESS</code> The optional value has been successfully set. <code>CVLS_MEM_NULL</code> The <code>cvode_mem</code> pointer is <code>NULL</code> . <code>CVLS_LMEM_NULL</code> The CVLS linear solver has not been initialized. <code>CVLS_SUNLS_FAIL</code> An error occurred when setting up the system matrix-times-vector routines in the <code>SUNLINSOL</code> object used by the CVLS interface.
Notes	<p>The default is to use an internal finite difference quotient for <code>jtimes</code> and to omit <code>jtsetup</code>. If <code>NULL</code> is passed to <code>jtimes</code>, these defaults are used. A user may specify non-<code>NULL</code> <code>jtimes</code> and <code>NULL</code> <code>jtsetup</code> inputs.</p> <p>This function must be called <i>after</i> the CVLS linear solver interface has been initialized through a call to <code>CVodeSetLinearSolver</code>.</p> <p>The function type <code>CVLSJacTimesSetupFn</code> is described in §4.6.7.</p> <p>The function type <code>CVLSJacTimesVecFn</code> is described in §4.6.6.</p> <p>The previous routine <code>CVSpilsSetJacTimes</code> 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.</p>

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

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

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

CVodeSetPreconditioner

Call	<code>flag = CVodeSetPreconditioner(cvode_mem, psetup, psolve);</code>
Description	The function <code>CVodeSetPreconditioner</code> specifies the preconditioner setup and solve functions.
Arguments	<p><code>cvode_mem</code> (void *) pointer to the CVOICE memory block.</p> <p><code>psetup</code> (CVLSPrecSetupFn) user-defined preconditioner setup function. Pass NULL if no setup is necessary.</p> <p><code>psolve</code> (CVLSPrecSolveFn) user-defined preconditioner solve function.</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>CVLS_SUCCESS</code> The optional values have been successfully set.</p> <p><code>CVLS_MEM_NULL</code> The <code>cvode_mem</code> pointer is NULL.</p> <p><code>CVLS_LMEM_NULL</code> The CVLS linear solver has not been initialized.</p> <p><code>CVLS_SUNLS_FAIL</code> An error occurred when setting up preconditioning in the SUNLINSOL object used by the CVLS interface.</p>
Notes	<p>The default is NULL for both arguments (i.e., no preconditioning).</p> <p>This function must be called <i>after</i> the CVLS linear solver interface has been initialized through a call to <code>CVodeSetLinearSolver</code>.</p> <p>The function type <code>CVLSPrecSolveFn</code> is described in §4.6.8.</p> <p>The function type <code>CVLSPrecSetupFn</code> is described in §4.6.9.</p> <p>The previous routine <code>CVSpilsSetPreconditioner</code> 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.</p>

CVodeSetEpsLin

Call	<code>flag = CVodeSetEpsLin(cvode_mem, eplifac);</code>
Description	The function <code>CVodeSetEpsLin</code> specifies the factor by which the Krylov linear solver's convergence test constant is reduced from the nonlinear solver test constant.
Arguments	<p><code>cvode_mem</code> (void *) pointer to the CVOICE memory block.</p> <p><code>eplifac</code> (realtype) linear convergence safety factor (≥ 0.0).</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>CVLS_SUCCESS</code> The optional value has been successfully set.</p> <p><code>CVLS_MEM_NULL</code> The <code>cvode_mem</code> pointer is NULL.</p> <p><code>CVLS_LMEM_NULL</code> The CVLS linear solver has not been initialized.</p> <p><code>CVLS_ILL_INPUT</code> The factor <code>eplifac</code> is negative.</p>
Notes	<p>The default value is 0.05.</p> <p>This function must be called <i>after</i> the CVLS linear solver interface has been initialized through a call to <code>CVodeSetLinearSolver</code>.</p> <p>If <code>eplifac= 0.0</code> is passed, the default value is used.</p> <p>The previous routine <code>CVSpilsSetEpsLin</code> 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.</p>

4.5.7.3 Rootfinding optional input functions

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

CVodeSetRootDirection

Call	<code>flag = CVodeSetRootDirection(cvode_mem, rootdir);</code>
Description	The function <code>CVodeSetRootDirection</code> specifies the direction of zero-crossings to be located and returned.
Arguments	<p><code>cvode_mem</code> (void *) pointer to the CVODE memory block.</p> <p><code>rootdir</code> (int *) state array of length <code>nrtfn</code>, the number of root functions g_i, as specified in the call to the function <code>CVodeRootInit</code>. A value of 0 for <code>rootdir[i]</code> 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.</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>CV_SUCCESS</code> The optional value has been successfully set.</p> <p><code>CV_MEM_NULL</code> The <code>cvode_mem</code> pointer is NULL.</p> <p><code>CV_ILL_INPUT</code> rootfinding has not been activated through a call to <code>CVodeRootInit</code>.</p>
Notes	The default behavior is to monitor for both zero-crossing directions.

CVodeSetNoInactiveRootWarn

Call	<code>flag = CVodeSetNoInactiveRootWarn(cvode_mem);</code>
Description	The function <code>CVodeSetNoInactiveRootWarn</code> disables issuing a warning if some root function appears to be identically zero at the beginning of the integration.
Arguments	<code>cvode_mem</code> (void *) pointer to the CVODE memory block.
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>CV_SUCCESS</code> The optional value has been successfully set.</p> <p><code>CV_MEM_NULL</code> The <code>cvode_mem</code> pointer is NULL.</p>
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.

4.5.8 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	<code>flag = CVodeGetDky(cvode_mem, t, k, dky);</code>
Description	The function <code>CVodeGetDky</code> computes the k -th derivative of the function y at time t , i.e. $d^{(k)}y/dt^{(k)}(t)$, where $t_n - h_u \leq t \leq t_n$, 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 $k = 0, 1, \dots, q_u$, where q_u is the current order (optional output <code>qlast</code>).
Arguments	<p><code>cvode_mem</code> (void *) pointer to the CVODE memory block.</p> <p><code>t</code> (realtype) the value of the independent variable at which the derivative is to be evaluated.</p>

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 <ul style="list-style-type: none"> CV_SUCCESS CVodeGetDky succeeded. CV_BAD_K k is not in the range $0, 1, \dots, q_u$. 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.

4.5.9 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.

4.5.9.1 SUNDIALS version information

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

SUNDIALSGetVersion	
Call	<code>flag = SUNDIALSGetVersion(version, len);</code>
Description	The function SUNDIALSGetVersion fills a character array with SUNDIALS version information.
Arguments	version (char *) character array to hold the SUNDIALS version information. len (int) allocated length of the version character array.
Return value	If successful, SUNDIALSGetVersion returns 0 and version contains the SUNDIALS version information. Otherwise, it returns -1 and version is not set (the input character array is too short).
Notes	A string of 25 characters should be sufficient to hold the version information. Any trailing characters in the version array are removed.

Table 4.3: Optional outputs from CVODE, CVLS, and CVDIAG

Optional output	Function name
CVODE main solver	
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
Last return from a linear solver function	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	<code>flag = SUNDIALSGetVersionNumber(&major, &minor, &patch, label, len);</code>
Description	The function <code>SUNDIALSGetVersionNumber</code> 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 <code>label</code> character array.
Return value	If successful, <code>SUNDIALSGetVersionNumber</code> returns 0 and the <code>major</code> , <code>minor</code> , <code>patch</code> , and <code>label</code> values are set. Otherwise, it returns <code>-1</code> 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 <code>label</code> . Any trailing characters in the <code>label</code> array are removed.

4.5.9.2 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	<code>flag = CVodeGetWorkSpace(cvode_mem, &lenrw, &leniw);</code>
Description	The function <code>CVodeGetWorkSpace</code> returns the CVODE real and integer workspace sizes.
Arguments	cvode_mem (void *) pointer to the CVODE memory block. lenrw (long int) the number of <code>realtype</code> values in the CVODE workspace. leniw (long int) the number of integer values in the CVODE workspace.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of CV_SUCCESS The optional output values have been successfully set. CV_MEM_NULL The <code>cvode_mem</code> pointer is <code>NULL</code> .
Notes	In terms of the problem size N , the maximum method order <code>maxord</code> , and the number <code>nrtfn</code> of root functions (see §4.5.5), the actual size of the real workspace, in <code>realtype</code> words, is given by the following:

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

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: $\text{leniw} = 40 + (\text{maxord}+5) * N_i + \text{nrtfn}$;
- using `CVodeSVtolerances`: $\text{leniw} = \text{leniw} + N_i$;

- with constraint checking: $\text{lenrw} = \text{lenrw} + N_i$;

where N_i is the number of integer words in one `N_Vector` (= 1 for `NVECTOR_SERIAL` and $2 \cdot \text{npes}$ 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: $\text{lenrw} = 96 + 17N$ and $\text{leniw} = 57$
- For the BDF method: $\text{lenrw} = 96 + 10N$ and $\text{leniw} = 50$

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.

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.

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

`CV_SUCCESS` The optional output value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is NULL.

CVoiceGetNumErrTestFails

Call `flag = CVoiceGetNumErrTestFails(cvoice_mem, &netfails);`

Description The function `CVoiceGetNumErrTestFails` returns the number of local error test failures that have occurred.

Arguments `cvoice_mem` (void *) pointer to the CVOICE 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 `cvoice_mem` pointer is NULL.

CVoiceGetLastOrder

Call `flag = CVoiceGetLastOrder(cvoice_mem, &qlast);`

Description The function `CVoiceGetLastOrder` returns the integration method order used during the last internal step.

Arguments `cvoice_mem` (void *) pointer to the CVOICE 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 `cvoice_mem` pointer is NULL.

CVoiceGetCurrentOrder

Call `flag = CVoiceGetCurrentOrder(cvoice_mem, &qcur);`

Description The function `CVoiceGetCurrentOrder` returns the integration method order to be used on the next internal step.

Arguments `cvoice_mem` (void *) pointer to the CVOICE 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 `cvoice_mem` pointer is NULL.

CVoiceGetLastStep

Call `flag = CVoiceGetLastStep(cvoice_mem, &hlast);`

Description The function `CVoiceGetLastStep` returns the integration step size taken on the last internal step.

Arguments `cvoice_mem` (void *) pointer to the CVOICE 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 `cvoice_mem` pointer is NULL.

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`.

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 ($h_{\min} \leq h_0 \leq h_{\max}$), or to satisfy the local error test condition.

CVodeGetCurrentTime

Call `flag = CVodeGetCurrentTime(cvode_mem, &tcure);`

Description The function `CVodeGetCurrentTime` returns the current internal time reached by the solver.

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

CVodeGetNumStabLimOrderReds

Call `flag = CVodeGetNumStabLimOrderReds(cvode_mem, &nsred);`

Description The function `CVodeGetNumStabLimOrderReds` returns the number of order reductions dictated by the BDF stability limit detection algorithm (see §2.3).

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`nsred` (`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 `nsred = 0`.

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.

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 (2.7).

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`.

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]`.

CVodeGetIntegratorStats

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

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.

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.

CVodeGetNumNonlinSolvConvFails

Call `flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &nncfails);`
 Description The function `CVodeGetNumNonlinSolvConvFails` returns the number of nonlinear convergence 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.

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 `ccode_mem` pointer is NULL.
 CV_MEM_FAIL The SUNNONLINSOL module is NULL.


CVodeGetReturnFlagName

Call `name = CVodeGetReturnFlagName(flag);`
 Description The function `CVodeGetReturnFlagName` returns the name of the CVODE constant corresponding 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.

4.5.9.3 Rootfinding optional output functions

There are two optional output functions associated with rootfinding.

CVodeGetRootInfo

Call `flag = CVodeGetRootInfo(ccode_mem, rootsfound);`
 Description The function `CVodeGetRootInfo` returns an array showing which functions were found to have a root.
 Arguments `ccode_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 $i = 0, \dots, \text{nrtfn}-1$, `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 `ccode_mem` pointer is NULL.
 Notes Note that, for the components g_i for which a root was found, the sign of `rootsfound[i]` indicates the direction of zero-crossing. A value of $+1$ indicates that g_i is increasing, while a value of -1 indicates a decreasing g_i .
 The user must allocate memory for the vector `rootsfound`.

CVodeGetNumGEvals

Call `flag = CVodeGetNumGEvals(ccode_mem, &ngevals);`
 Description The function `CVodeGetNumGEvals` returns the cumulative number of calls made to the user-supplied root function g .
 Arguments `ccode_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 `ccode_mem` pointer is NULL.

4.5.9.4 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`).

CVodeGetLinWorkSpace

Call	<code>flag = CVodeGetLinWorkSpace(cvode_mem, &lenrwLS, &leniwLS);</code>
Description	The function <code>CVodeGetLinWorkSpace</code> returns the sizes of the real and integer workspaces used by the CVLS linear solver interface.
Arguments	<p><code>cvode_mem</code> (<code>void *</code>) pointer to the CVODE memory block.</p> <p><code>lenrwLS</code> (<code>long int</code>) the number of <code>realtype</code> values in the CVLS workspace.</p> <p><code>leniwLS</code> (<code>long int</code>) the number of integer values in the CVLS workspace.</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>CVLS_SUCCESS</code> The optional output values have been successfully set.</p> <p><code>CVLS_MEM_NULL</code> The <code>cvode_mem</code> pointer is NULL.</p> <p><code>CVLS_LMEM_NULL</code> The CVLS linear solver has not been initialized.</p>
Notes	<p>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.</p> <p>The previous routines <code>CVDlsGetWorkspace</code> and <code>CVSpilsGetWorkspace</code> are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to the new routine name soon.</p>

CVodeGetNumJacEvals

Call	<code>flag = CVodeGetNumJacEvals(cvode_mem, &njevals);</code>
Description	The function <code>CVodeGetNumJacEvals</code> returns the number of calls made to the CVLS Jacobian approximation function.
Arguments	<p><code>cvode_mem</code> (<code>void *</code>) pointer to the CVODE memory block.</p> <p><code>njevals</code> (<code>long int</code>) the number of calls to the Jacobian function.</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of</p> <p><code>CVLS_SUCCESS</code> The optional output value has been successfully set.</p> <p><code>CVLS_MEM_NULL</code> The <code>cvode_mem</code> pointer is NULL.</p> <p><code>CVLS_LMEM_NULL</code> The CVLS linear solver has not been initialized.</p>
Notes	The previous routine <code>CVDlsGetNumJacEvals</code> 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.

CVodeGetNumLinRhsEvals

Call	<code>flag = CVodeGetNumLinRhsEvals(cvode_mem, &nfevalsLS);</code>
Description	The function <code>CVodeGetNumLinRhsEvals</code> 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	<code>cvode_mem</code> (<code>void *</code>) pointer to the CVMODE memory block. <code>nfevalsLS</code> (<code>long int</code>) the number of calls made to the user-supplied right-hand side function.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of <code>CVLS_SUCCESS</code> The optional output value has been successfully set. <code>CVLS_MEM_NULL</code> The <code>cvode_mem</code> pointer is <code>NULL</code> . <code>CVLS_LMEM_NULL</code> The CVLS linear solver has not been initialized.
Notes	The value <code>nfevalsLS</code> is incremented only if one of the default internal difference quotient functions is used. The previous routines <code>CVDlsGetNumRhsEvals</code> and <code>CVSpilsGetNumRhsEvals</code> are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

CVodeGetNumLinIters

Call	<code>flag = CVodeGetNumLinIters(cvode_mem, &nliters);</code>
Description	The function <code>CVodeGetNumLinIters</code> returns the cumulative number of linear iterations.
Arguments	<code>cvode_mem</code> (<code>void *</code>) pointer to the CVMODE memory block. <code>nliters</code> (<code>long int</code>) the current number of linear iterations.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of <code>CVLS_SUCCESS</code> The optional output value has been successfully set. <code>CVLS_MEM_NULL</code> The <code>cvode_mem</code> pointer is <code>NULL</code> . <code>CVLS_LMEM_NULL</code> The CVLS linear solver has not been initialized.
Notes	The previous routine <code>CVSpilsGetNumLinIters</code> 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.

CVodeGetNumLinConvFails

Call	<code>flag = CVodeGetNumLinConvFails(cvode_mem, &nlcfails);</code>
Description	The function <code>CVodeGetNumLinConvFails</code> returns the cumulative number of linear convergence failures.
Arguments	<code>cvode_mem</code> (<code>void *</code>) pointer to the CVMODE memory block. <code>nlcfails</code> (<code>long int</code>) the current number of linear convergence failures.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of <code>CVLS_SUCCESS</code> The optional output value has been successfully set. <code>CVLS_MEM_NULL</code> The <code>cvode_mem</code> pointer is <code>NULL</code> . <code>CVLS_LMEM_NULL</code> The CVLS linear solver has not been initialized.
Notes	The previous routine <code>CVSpilsGetNumConvFails</code> 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.

CVodeGetNumPrecEvals

Call	<code>flag = CVodeGetNumPrecEvals(cvode_mem, &npevals);</code>
Description	The function <code>CVodeGetNumPrecEvals</code> returns the number of preconditioner evaluations, i.e., the number of calls made to <code>psetup</code> with <code>jok = SUNFALSE</code> .
Arguments	<code>cvode_mem</code> (<code>void *</code>) pointer to the CVODE memory block. <code>npevals</code> (<code>long int</code>) the current number of calls to <code>psetup</code> .
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of <code>CVLS_SUCCESS</code> The optional output value has been successfully set. <code>CVLS_MEM_NULL</code> The <code>cvode_mem</code> pointer is <code>NULL</code> . <code>CVLS_LMEM_NULL</code> The CVLS linear solver has not been initialized.
Notes	The previous routine <code>CVSpilsGetNumPrecEvals</code> 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.

CVodeGetNumPrecSolves

Call	<code>flag = CVodeGetNumPrecSolves(cvode_mem, &npsolves);</code>
Description	The function <code>CVodeGetNumPrecSolves</code> returns the cumulative number of calls made to the preconditioner solve function, <code>psolve</code> .
Arguments	<code>cvode_mem</code> (<code>void *</code>) pointer to the CVODE memory block. <code>npsolves</code> (<code>long int</code>) the current number of calls to <code>psolve</code> .
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of <code>CVLS_SUCCESS</code> The optional output value has been successfully set. <code>CVLS_MEM_NULL</code> The <code>cvode_mem</code> pointer is <code>NULL</code> . <code>CVLS_LMEM_NULL</code> The CVLS linear solver has not been initialized.
Notes	The previous routine <code>CVSpilsGetNumPrecSolves</code> 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.

CVodeGetNumJTSetupEvals

Call	<code>flag = CVodeGetNumJTSetupEvals(cvode_mem, &njtsetup);</code>
Description	The function <code>CVodeGetNumJTSetupEvals</code> returns the cumulative number of calls made to the Jacobian-vector setup function <code>jtsetup</code> .
Arguments	<code>cvode_mem</code> (<code>void *</code>) pointer to the CVODE memory block. <code>njtsetup</code> (<code>long int</code>) the current number of calls to <code>jtsetup</code> .
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of <code>CVLS_SUCCESS</code> The optional output value has been successfully set. <code>CVLS_MEM_NULL</code> The <code>cvode_mem</code> pointer is <code>NULL</code> . <code>CVLS_LMEM_NULL</code> The CVLS linear solver has not been initialized.
Notes	The previous routine <code>CVSpilsGetNumJTSetupEvals</code> 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.

CVodeGetNumJtimesEvals

Call	<code>flag = CVodeGetNumJtimesEvals(cvode_mem, &njvevals);</code>
Description	The function <code>CVodeGetNumJtimesEvals</code> returns the cumulative number of calls made to the Jacobian-vector function <code>jtimes</code> .
Arguments	<code>cvode_mem</code> (void *) pointer to the CVOICE memory block. <code>njvevals</code> (long int) the current number of calls to <code>jtimes</code> .
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of <code>CVLS_SUCCESS</code> The optional output value has been successfully set. <code>CVLS_MEM_NULL</code> The <code>cvode_mem</code> pointer is <code>NULL</code> . <code>CVLS_LMEM_NULL</code> The CVLS linear solver has not been initialized.
Notes	The previous routine <code>CVSpilsGetNumJtimesEvals</code> 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.

CVodeGetLastLinFlag

Call	<code>flag = CVodeGetLastLinFlag(cvode_mem, &lsflag);</code>
Description	The function <code>CVodeGetLastLinFlag</code> returns the last return value from a CVLS routine.
Arguments	<code>cvode_mem</code> (void *) pointer to the CVOICE memory block. <code>lsflag</code> (long int) the value of the last return flag from a CVLS function.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of <code>CVLS_SUCCESS</code> The optional output value has been successfully set. <code>CVLS_MEM_NULL</code> The <code>cvode_mem</code> pointer is <code>NULL</code> . <code>CVLS_LMEM_NULL</code> The CVLS linear solver has not been initialized.
Notes	<p>If the CVLS setup function failed (i.e., <code>CVode</code> returned <code>CV_LSETUP_FAIL</code>) when using the <code>SUNLINSOL_DENSE</code> or <code>SUNLINSOL_BAND</code> modules, then the value of <code>lsflag</code> 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.</p> <p>If the CVLS setup function failed when using another <code>SUNLINSOL</code> module, then <code>lsflag</code> will be <code>SUNLS_PSET_FAIL_UNREC</code>, <code>SUNLS_ASET_FAIL_UNREC</code>, or <code>SUNLS_PACKAGE_FAIL_UNREC</code>.</p> <p>If the CVLS solve function failed (i.e., <code>CVode</code> returned <code>CV_LSOLVE_FAIL</code>), then <code>lsflag</code> contains the error return flag from the <code>SUNLINSOL</code> object, which will be one of: <code>SUNLS_MEM_NULL</code>, indicating that the <code>SUNLINSOL</code> memory is <code>NULL</code>; <code>SUNLS_ATIMES_FAIL_UNREC</code>, indicating an unrecoverable failure in the <code>Jv</code> function; <code>SUNLS_PSOLVE_FAIL_UNREC</code>, indicating that the preconditioner solve function <code>psolve</code> failed unrecoverably; <code>SUNLS_GS_FAIL</code>, indicating a failure in the Gram-Schmidt procedure (SPGMR and SPFGMR only); <code>SUNLS_QRSOL_FAIL</code>, indicating that the matrix R was found to be singular during the QR solve phase (SPGMR and SPFGMR only); or <code>SUNLS_PACKAGE_FAIL_UNREC</code>, indicating an unrecoverable failure in an external iterative linear solver package.</p> <p>The previous routines <code>CVDlsGetLastFlag</code> and <code>CVSpilsGetLastFlag</code> are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to the new routine name soon.</p>

CVodeGetLinReturnFlagName

Call	<code>name = CVodeGetLinReturnFlagName(lsflag);</code>
Description	The function <code>CVodeGetLinReturnFlagName</code> returns the name of the CVLS constant corresponding to <code>lsflag</code> .

Arguments	The only argument, of type <code>long int</code> , is a return flag from a CVLS function.
Return value	The return value is a string containing the name of the corresponding constant. If $1 \leq \text{lsflag} \leq N$ (LU factorization failed), this routine returns “NONE”.
Notes	The previous routines <code>CVDlsGetReturnFlagName</code> and <code>CVSpilsGetReturnFlagName</code> are now wrappers for this routine, and may still be used for backward-compatibility. However, these will be deprecated in future releases, so we recommend that users transition to the new routine name soon.

4.5.9.5 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	<code>flag = CVDiagGetWorkSpace(cvode_mem, &lenrwLS, &leniwLS);</code>
Description	The function <code>CVDiagGetWorkSpace</code> returns the CVDIAG real and integer workspace sizes.
Arguments	<code>cvode_mem</code> (<code>void *</code>) pointer to the CVODE memory block. <code>lenrwLS</code> (<code>long int</code>) the number of <code>realtype</code> values in the CVDIAG workspace. <code>leniwLS</code> (<code>long int</code>) the number of integer values in the CVDIAG workspace.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of <code>CVDIAG_SUCCESS</code> The optional output values have been successfully set. <code>CVDIAG_MEM_NULL</code> The <code>cvode_mem</code> pointer is NULL. <code>CVDIAG_LMEM_NULL</code> 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 $3N$ <code>realtype</code> words.

CVDiagGetNumRhsEvals	
Call	<code>flag = CVDiagGetNumRhsEvals(cvode_mem, &nfevalsLS);</code>
Description	The function <code>CVDiagGetNumRhsEvals</code> returns the number of calls made to the user-supplied right-hand side function due to the finite difference Jacobian approximation.
Arguments	<code>cvode_mem</code> (<code>void *</code>) pointer to the CVODE memory block. <code>nfevalsLS</code> (<code>long int</code>) the number of calls made to the user-supplied right-hand side function.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of <code>CVDIAG_SUCCESS</code> The optional output value has been successfully set. <code>CVDIAG_MEM_NULL</code> The <code>cvode_mem</code> pointer is NULL. <code>CVDIAG_LMEM_NULL</code> 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 <code>CVodeGetNumLinSolvSetups</code>).

CVDiagGetLastFlag

Call	<code>flag = CVDiagGetLastFlag(cvode_mem, &lsflag);</code>
Description	The function <code>CVDiagGetLastFlag</code> returns the last return value from a CVDIAG routine.
Arguments	<code>cvode_mem</code> (<code>void *</code>) pointer to the CVODE memory block. <code>lsflag</code> (<code>long int</code>) the value of the last return flag from a CVDIAG function.
Return value	The return value <code>flag</code> (of type <code>int</code>) is one of <code>CVDIAG_SUCCESS</code> The optional output value has been successfully set. <code>CVDIAG_MEM_NULL</code> The <code>cvode_mem</code> pointer is <code>NULL</code> . <code>CVDIAG_LMEM_NULL</code> The CVDIAG linear solver has not been initialized.
Notes	If the CVDIAG setup function failed (<code>CVode</code> returned <code>CV_LSETUP_FAIL</code>), the value of <code>lsflag</code> is equal to <code>CVDIAG_INV_FAIL</code> , indicating that a diagonal element with value zero was encountered. The same value is also returned if the CVDIAG solve function failed (<code>CVode</code> returned <code>CV_LSOLVE_FAIL</code>).

CVDiagGetReturnFlagName

Call	<code>name = CVDiagGetReturnFlagName(lsflag);</code>
Description	The function <code>CVDiagGetReturnFlagName</code> returns the name of the CVDIAG constant corresponding to <code>lsflag</code> .
Arguments	The only argument, of type <code>long int</code> , is a return flag from a CVDIAG function.
Return value	The return value is a string containing the name of the corresponding constant.

4.5.10 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 `lmm` 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	<code>flag = CVodeReInit(cvode_mem, t0, y0);</code>
Description	The function <code>CVodeReInit</code> provides required problem specifications and reinitializes CVODE.
Arguments	<p><code>cvode_mem</code> (<code>void *</code>) pointer to the CVODE memory block.</p> <p><code>t0</code> (<code>realtype</code>) is the initial value of t.</p> <p><code>y0</code> (<code>N_Vector</code>) is the initial value of y.</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) will be one of the following:</p> <p><code>CV_SUCCESS</code> The call to <code>CVodeReInit</code> was successful.</p> <p><code>CV_MEM_NULL</code> The CVODE memory block was not initialized through a previous call to <code>CVodeCreate</code>.</p> <p><code>CV_NO_MALLOC</code> Memory space for the CVODE memory block was not allocated through a previous call to <code>CVodeInit</code>.</p> <p><code>CV_ILL_INPUT</code> An input argument to <code>CVodeReInit</code> has an illegal value.</p>
Notes	If an error occurred, <code>CVodeReInit</code> also sends an error message to the error handler function.

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	
Definition	<pre>typedef int (*CVRhsFn)(realtype t, N_Vector y, N_Vector ydot, void *user_data);</pre>
Purpose	This function computes the ODE right-hand side for a given value of the independent variable t and state vector y .
Arguments	<p><code>t</code> is the current value of the independent variable.</p> <p><code>y</code> is the current value of the dependent variable vector, $y(t)$.</p> <p><code>ydot</code> is the output vector $f(t, y)$.</p> <p><code>user_data</code> is the <code>user_data</code> pointer passed to <code>CVodeSetUserData</code>.</p>
Return value	A <code>CVRhsFn</code> 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 <code>CV_RHSFUNC_FAIL</code> is returned).
Notes	<p>Allocation of memory for <code>ydot</code> is handled within CVODE.</p> <p>A recoverable failure error return from the <code>CVRhsFn</code> is typically used to flag a value of the dependent variable y that is “illegal” in some way (e.g., negative where only a non-negative value is physically meaningful). If such a return is made, CVODE will attempt to recover (possibly repeating the nonlinear solve, or reducing the step size) in order to avoid this recoverable error return.</p>

Notes	<p>Allocation of memory for <code>ewt</code> is handled within <code>CVODE</code>.</p> <p>The error weight vector must have all components positive. It is the user's responsibility to perform this test and return <code>-1</code> if it is not satisfied.</p>
-------	--



4.6.4 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	<pre>typedef int (*CVRootFn)(realtype t, N_Vector y, realtype *gout, void *user_data);</pre>
Purpose	This function implements a vector-valued function $g(t, y)$ such that the roots of the <code>nrtfn</code> components $g_i(t, y)$ are sought.
Arguments	<p><code>t</code> is the current value of the independent variable.</p> <p><code>y</code> is the current value of the dependent variable vector, $y(t)$.</p> <p><code>gout</code> is the output array, of length <code>nrtfn</code>, with components $g_i(t, y)$.</p> <p><code>user_data</code> is a pointer to user data, the same as the <code>user_data</code> parameter passed to <code>CVodeSetUserData</code>.</p>
Return value	A <code>CVRootFn</code> should return 0 if successful or a non-zero value if an error occurred (in which case the integration is halted and <code>CVode</code> returns <code>CV_RTFUNC_FAIL</code>).
Notes	Allocation of memory for <code>gout</code> is automatically handled within <code>CVODE</code> .

4.6.5 Jacobian construction (matrix-based linear solvers)

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

CVLsJacFn	
Definition	<pre>typedef (*CVLsJacFn)(realtype t, N_Vector y, N_Vector fy, SUNMatrix Jac, void *user_data, N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);</pre>
Purpose	This function computes the Jacobian matrix $J = \partial f / \partial y$ (or an approximation to it).
Arguments	<p><code>t</code> is the current value of the independent variable.</p> <p><code>y</code> is the current value of the dependent variable vector, namely the predicted value of $y(t)$.</p> <p><code>fy</code> is the current value of the vector $f(t, y)$.</p> <p><code>Jac</code> is the output Jacobian matrix (of type <code>SUNMatrix</code>).</p> <p><code>user_data</code> is a pointer to user data, the same as the <code>user_data</code> parameter passed to <code>CVodeSetUserData</code>.</p> <p><code>tmp1</code> <code>tmp2</code> <code>tmp3</code> are pointers to memory allocated for variables of type <code>N_Vector</code> which can be used by a <code>CVLsJacFn</code> function as temporary storage or work space.</p>
Return value	A <code>CVLsJacFn</code> should return 0 if successful, a positive value if a recoverable error occurred (in which case <code>CVODE</code> will attempt to correct, while <code>CVLS</code> sets <code>last_flag</code> to <code>CVLS_JACFUNC_RECVR</code>), or a negative value if it failed unrecoverably (in which case the integration is halted, <code>CVode</code> returns <code>CV_LSETUP_FAIL</code> and <code>CVLS</code> sets <code>last_flag</code> to <code>CVLS_JACFUNC_UNRECVR</code>).

Notes

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

Prior to calling the user-supplied Jacobian function, the Jacobian matrix $J(t, y)$ is zeroed out, so only nonzero elements need to be loaded into `Jac`.

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 §4.5.9.2. 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 `SUNMATRIX_DENSE` type. `SM_ELEMENT_D(J, i, j)` references the (i, j) -th element of the dense matrix `Jac` (with $i, j = 0 \dots N - 1$). This macro is meant for small problems for which efficiency of access is not a major concern. Thus, in terms of the indices m and n ranging from 1 to N , the Jacobian element $J_{m,n}$ can be set using the statement `SM_ELEMENT_D(J, m-1, n-1) = J_{m,n}`. Alternatively, `SM_COLUMN_D(J, j)` returns a pointer to the first element of the j -th column of `Jac` (with $j = 0 \dots N - 1$), and the elements of the j -th column can then be accessed using ordinary array indexing. Consequently, $J_{m,n}$ can be loaded using the statements `col_n = SM_COLUMN_D(J, n-1); col_n[m-1] = J_{m,n}`. For large problems, it is more efficient to use `SM_COLUMN_D` than to use `SM_ELEMENT_D`. Note that both of these macros number rows and columns starting from 0. The `SUNMATRIX_DENSE` type and accessor macros are documented in §7.1.

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 $-\text{mupper} \leq m-n \leq \text{mlower}$. Alternatively, `SM_COLUMN_B(J, j)` returns a pointer to the diagonal element of the j -th column of `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 `SUNMATRIX_BAND`. The array `col_n` can be indexed from $-\text{mupper}$ to `mlower`. For large problems, it is more efficient to use `SM_COLUMN_B` and `SM_COLUMN_ELEMENT_B` than to use the `SM_ELEMENT_B` macro. As in the dense case, these macros all number rows and columns starting from 0. The `SUNMATRIX_BAND` type and accessor macros are documented in §7.2.

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 §7.3.

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.

4.6.6 Jacobian-vector product (matrix-free linear solvers)

If a matrix-free linear solver is to be used (i.e., a `NULL`-valued `SUNMATRIX` was supplied to `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	<pre>typedef int (*CVLsJacTimesVecFn)(N_Vector v, N_Vector Jv, realtype t, N_Vector y, N_Vector fy, void *user_data, N_Vector tmp);</pre>	
Purpose	This function computes the product $Jv = (\partial f / \partial y)v$ (or an approximation to it).	
Arguments	<code>v</code>	is the vector by which the Jacobian must be multiplied.
	<code>Jv</code>	is the output vector computed.
	<code>t</code>	is the current value of the independent variable.
	<code>y</code>	is the current value of the dependent variable vector.
	<code>fy</code>	is the current value of the vector $f(t, y)$.
	<code>user_data</code>	is a pointer to user data, the same as the <code>user_data</code> parameter passed to <code>CVodeSetUserData</code> .
	<code>tmp</code>	is a pointer to memory allocated for a variable of type <code>N_Vector</code> 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 <i>current</i> value of J , i.e. as evaluated at the current (t, y) .	
	If the user's <code>CVLsJacTimesVecFn</code> 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 <code>cv_mem</code> to <code>user_data</code> and then use the <code>CVodeGet*</code> functions described in §4.5.9.2. The unit roundoff can be accessed as <code>UNIT_ROUNDOFF</code> defined in <code>sundials.types.h</code> .	
	The previous function type <code>CVSpilsJacTimesVecFn</code> is identical to <code>CVLsJacTimesVecFn</code> , and may still be used for backward-compatibility. However, this will be deprecated in future releases, so we recommend that users transition to the new function type name soon.	

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

If the user's Jacobian-times-vector 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:

CVLsJacTimesSetupFn

Definition	<code>typedef int (*CVLsJacTimesSetupFn)(realtype t, N_Vector y, N_Vector fy, void *user_data);</code>
Purpose	This function preprocesses and/or evaluates Jacobian-related data needed by the Jacobian-times-vector routine.
Arguments	<p><code>t</code> is the current value of the independent variable.</p> <p><code>y</code> is the current value of the dependent variable vector.</p> <p><code>fy</code> is the current value of the vector $f(t, y)$.</p> <p><code>user_data</code> is a pointer to user data, the same as the <code>user_data</code> parameter passed to <code>CVodeSetUserData</code>.</p>
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	<p>Each call to the Jacobian-vector setup function is preceded by a call to the <code>CVRhsFn</code> user function with the same <code>(t,y)</code> arguments. Thus, the setup function can use any auxiliary data that is computed and saved during the evaluation of the ODE right-hand side.</p> <p>If the user's <code>CVLsJacTimesSetupFn</code> 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 <code>cv_mem</code> to <code>user_data</code> and then use the <code>CVodeGet*</code> functions described in §4.5.9.2. The unit roundoff can be accessed as <code>UNIT_ROUNDOFF</code> defined in <code>sundials.types.h</code>.</p> <p>The previous function type <code>CVSpilsJacTimesSetupFn</code> is identical to <code>CVLsJacTimesSetupFn</code>, 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.</p>

4.6.8 Preconditioner solve (iterative linear solvers)

If a user-supplied preconditioner is to be used with a SUNLINSOL solver module, then the user must provide a function to solve the linear system $Pz = r$, where P 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	<code>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);</code>
Purpose	This function solves the preconditioned system $Pz = r$.
Arguments	<p><code>t</code> is the current value of the independent variable.</p> <p><code>y</code> is the current value of the dependent variable vector.</p> <p><code>fy</code> is the current value of the vector $f(t, y)$.</p> <p><code>r</code> is the right-hand side vector of the linear system.</p> <p><code>z</code> is the computed output vector.</p> <p><code>gamma</code> is the scalar γ appearing in the matrix given by $M = I - \gamma J$.</p> <p><code>delta</code> 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 <code>delta</code> in the weighted l_2 norm, i.e., $\sqrt{\sum_i (Res_i \cdot ewt_i)^2} < \text{delta}$. To obtain the <code>N_Vector</code> <code>ewt</code>, call <code>CVodeGetErrWeights</code> (see §4.5.9.2).</p>

1r	is an input flag indicating whether the preconditioner solve function is to use the left preconditioner (1r = 1) or the right preconditioner (1r = 2);
user_data	is a pointer to user data, the same as the user_data parameter passed to the function CNodeSetUserData .
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).
Notes	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.

4.6.9 Preconditioner setup (iterative linear solvers)

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

CVLsPrecSetupFn

Definition	<pre>typedef int (*CVLsPrecSetupFn)(realtype t, N_Vector y, N_Vector fy, booleantype jok, booleantype *jcurPtr, realtype gamma, void *user_data);</pre>	
Purpose	This function preprocesses and/or evaluates Jacobian-related data needed by the preconditioner.	
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 $f(t, y)$.
	jok	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 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 = SUNFALSE .
	jcurPtr	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 CNodeSetUserData .
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).	
Notes	<p>The operations performed by this function might include forming a crude approximate Jacobian and performing an LU factorization of the resulting approximation to $M = I - \gamma J$.</p> <p>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</p>	

use any auxiliary data that is computed and saved during the evaluation of the ODE right-hand side.

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 §4.5.9.2. 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.

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 `m1`, respectively) and call

```
flag = CVBandPrecInit(cvode_mem, N, mu, m1);
```

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, m1);`

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.
`m1` (`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 $-m1 \leq j - i \leq mu$.

The following three optional output functions are available for use with the CVBANDPRE module:

CVBandPrecGetWorkSpace

Call	<code>flag = CVBandPrecGetWorkSpace(cvode_mem, &lenrwBP, &leniwBP);</code>
Description	The function <code>CVBandPrecGetWorkSpace</code> returns the sizes of the CVBANDPRE real and integer workspaces.
Arguments	<p><code>cvode_mem</code> (void *) pointer to the CVOICE memory block.</p> <p><code>lenrwBP</code> (long int) the number of realtype values in the CVBANDPRE workspace.</p> <p><code>leniwBP</code> (long int) the number of integer values in the CVBANDPRE workspace.</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of:</p> <p><code>CVLS_SUCCESS</code> The optional output values have been successfully set.</p> <p><code>CVLS_PMEM_NULL</code> The CVBANDPRE preconditioner has not been initialized.</p>
Notes	<p>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).</p> <p>The workspaces referred to here exist in addition to those given by the corresponding function <code>CVodeGetLinWorkSpace</code>.</p>

CVBandPrecGetNumRhsEvals

Call	<code>flag = CVBandPrecGetNumRhsEvals(cvode_mem, &nfevalsBP);</code>
Description	The function <code>CVBandPrecGetNumRhsEvals</code> returns the number of calls made to the user-supplied right-hand side function for the finite difference banded Jacobian approximation used within the preconditioner setup function.
Arguments	<p><code>cvode_mem</code> (void *) pointer to the CVOICE memory block.</p> <p><code>nfevalsBP</code> (long int) the number of calls to the user right-hand side function.</p>
Return value	<p>The return value <code>flag</code> (of type <code>int</code>) is one of:</p> <p><code>CVLS_SUCCESS</code> The optional output value has been successfully set.</p> <p><code>CVLS_PMEM_NULL</code> The CVBANDPRE preconditioner has not been initialized.</p>
Notes	<p>The counter <code>nfevalsBP</code> is distinct from the counter <code>nfevalsLS</code> returned by the corresponding function <code>CVodeGetNumLinRhsEvals</code> and <code>nfevals</code> returned by <code>CVodeGetNumRhsEvals</code>. The total number of right-hand side function evaluations is the sum of all three of these counters.</p>

4.7.2 A parallel band-block-diagonal preconditioner module

A principal reason for using a parallel ODE solver such as CVOICE lies in the solution of partial differential equations (PDEs). Moreover, the use of a Krylov iterative method for the solution of many such problems is motivated by the nature of the underlying linear system of equations (2.5) that must be solved at each time step. The linear algebraic system is large, sparse, and structured. However, if a Krylov iterative method is to be effective in this setting, then a nontrivial preconditioner needs to be used. Otherwise, the rate of convergence of the Krylov iterative method is usually unacceptably slow. Unfortunately, an effective preconditioner tends to be problem-specific.

However, we have developed one type of preconditioner that treats a rather broad class of PDE-based problems. It has been successfully used for several realistic, large-scale problems [28] and is included in a software module within the CVOICE 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

$$g(t, y) = [g_1(t, \bar{y}_1), g_2(t, \bar{y}_2), \dots, g_M(t, \bar{y}_M)]^T \quad (4.1)$$

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

$$P = \text{diag}[P_1, P_2, \dots, P_M] \quad (4.2)$$

where

$$P_m \approx I - \gamma J_m \quad (4.3)$$

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

$$Px = b \quad (4.4)$$

reduces to solving each of the equations

$$P_m x_m = b_m \quad (4.5)$$

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 `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 `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 `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 `f`. The user is responsible for providing space (presumably within `user_data`) for components of `y` that are communicated between processes by `cfn`, and that are then used by `gloc`, which should not do any communication.

CVLocalFn

Definition	<pre>typedef int (*CVLocalFn)(sunindextype Nlocal, realtype t, N_Vector y, N_Vector glocal, void *user_data);</pre>		
Purpose	This <code>gloc</code> function computes $g(t, y)$. It loads the vector <code>glocal</code> as a function of <code>t</code> and <code>y</code> .		
Arguments	<code>Nlocal</code>	is the local vector length.	
	<code>t</code>	is the value of the independent variable.	
	<code>y</code>	is the dependent variable.	

	<code>glocal</code> is the output vector.
	<code>user_data</code> is a pointer to user data, the same as the <code>user_data</code> parameter passed to <code>CVodeSetUserData</code> .
Return value	A <code>CVLocalFn</code> 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 <code>CVode</code> returns <code>CV_LSETUP_FAIL</code>).
Notes	This function must assume that all interprocess communication of data needed to calculate <code>glocal</code> has already been done, and that this data is accessible within <code>user_data</code> . The case where g is mathematically identical to f is allowed.

CVCommFn

Definition	<pre>typedef int (*CVCommFn)(sunindextype Nlocal, realtype t, N_Vector y, void *user_data);</pre>
Purpose	This <code>cfn</code> function performs all interprocess communication necessary for the execution of the <code>gloc</code> function above, using the input vector <code>y</code> .
Arguments	<code>Nlocal</code> is the local vector length. <code>t</code> is the value of the independent variable. <code>y</code> is the dependent variable. <code>user_data</code> is a pointer to user data, the same as the <code>user_data</code> parameter passed to <code>CVodeSetUserData</code> .
Return value	A <code>CVCommFn</code> 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 <code>CVode</code> returns <code>CV_LSETUP_FAIL</code>).
Notes	The <code>cfn</code> function is expected to save communicated data in space defined within the data structure <code>user_data</code> . Each call to the <code>cfn</code> function is preceded by a call to the right-hand side function <code>f</code> with the same (<code>t</code> , <code>y</code>) arguments. Thus, <code>cfn</code> can omit any communication done by <code>f</code> if relevant to the evaluation of <code>glocal</code> . If all necessary communication was done in <code>f</code> , then <code>cfn = NULL</code> can be passed in the call to <code>CVBBDPrecInit</code> (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 §4.4 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

10. **Initialize the CVBBDPRE preconditioner module**

```
flag = CVBBDPrecInit(cvode_mem, local_N, mudq, mldq,
                    mukeep, mlkeep, dqrely, gloc, cfn);
```

11. Set optional inputs

12. Create nonlinear solver object

14. Set nonlinear solver optional inputs

15. Specify rootfinding problem

16. Advance solution in time

17. Get optional outputs

18. **Deallocate memory** for solution vector

19. Free solver memory

20. Free nonlinear solver memory

21. Free linear solver memory

22. Finalize MPI

CVBBDPrecInit

[illegible]

Description	The function <code>CVBBDPrecInit</code> initializes and allocates (internal) memory for the CVBBDPRE preconditioner.
-------------	--

Arguments	<code>ccode_mem</code> (void *) pointer to the CCODE memory block.
	<code>local_N</code> (<code>sunindextype</code>) local vector length.
	<code>mudq</code> (<code>sunindextype</code>) upper half-bandwidth to be used in the difference quotient Jacobian approximation.
	<code>mldq</code> (<code>sunindextype</code>) lower half-bandwidth to be used in the difference quotient Jacobian approximation.
	<code>mukeep</code> (<code>sunindextype</code>) upper half-bandwidth of the retained banded approximate Jacobian block.

mlkeep	(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 $\text{dqrely} = \sqrt{\text{unit roundoff}}$, which can be specified by passing $\text{dqrely} = 0.0$.
gloc	(CVLocalFn) the C function which computes the approximation $g(t, y) \approx f(t, y)$.
cfn	(CVCommFn) the optional C function which performs all interprocess communication required for the computation of $g(t, y)$.

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

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 g when smaller values may provide a greater efficiency.

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

For all four half-bandwidths, the values need not be the same on every processor.

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. mudq (sunindextype) upper half-bandwidth to be used in the difference quotient Jacobian approximation. mldq (sunindextype) lower half-bandwidth to be used in the difference quotient Jacobian approximation. dqrely (realtype) the relative increment in components of y used in the difference quotient approximations. The default is $\text{dqrely} = \sqrt{\text{unit roundoff}}$, which can be specified by passing $\text{dqrely} = 0.0$.

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.

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.
`lenrwBBDP` (long int) local number of **realtype** values in the 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 allocated within the CVBBDPRE module (the banded matrix approximation, banded SUNLINSOL object, temporary vectors). These values are local to each process.

The workspaces referred to here exist in addition to those given by the corresponding function `CVodeGetLinWorkSpace`.

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.

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 §4.5.9).

Chapter 5

FCVODE, an Interface Module for FORTRAN Applications

The FCVODE interface module is a package of C functions which support the use of the CVODE solver, 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. This package provides the necessary interface to CVODE for all supplied serial and parallel NVECTOR implementations.

5.1 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 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

5.3 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 `CNodeReInit`.
- FCVSETIIN and FCVSETRIN interface to `CNodeSet*` functions.
- FCVEWTSET interfaces to `CNodeWftolerances`.
- FCVODE interfaces to `CNode`, `CNodeGet*` functions, and to the optional output functions for the selected linear solver module.
- FCVDKY interfaces to the interpolated output function `CNodeGetDky`.
- FCVGETERWEIGHTS interfaces to `CNodeGetErrWeights`.
- FCVGESTESTLOCALERR interfaces to `CNodeGetEstLocalErrors`.
- FCVFREE interfaces to `CNodeFree`.
- Interface to the linear solver interfaces
 - FCVLSINIT interfaces to `CNodeSetLinearSolver`.
 - FCVLSSETEPSLIN interfaces to `CNodeSetEpsLin`.
 - FCVLSSETJAC interfaces to `CNodeSetJacTimes`.
 - FCVLSSETPREC interfaces to `CNodeSetPreconditioner`.
 - FCVDENSESETJAC interfaces to `CNodeSetJacFn`.
 - FCVBANDSETJAC interfaces to `CNodeSetJacFn`.
 - FCVSPARSESETJAC interfaces to `CNodeSetJacFn`.
 - 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 (FORTRAN, user-supplied)	CVODE function (C, interface)	CVODE type of interface function
FCVFUN	FCVf	CVRhsFn
FCVEWT	FCVEwtSet	CVEwtFn
FCVDJAC	FCVDenseJac	CVLsJacFn
FCVBJAC	FCVBandJac	CVLsJacFn
FCVSPJAC	FCVSparsJac	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.

5.4 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 FCFUN(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 FCFMALLOC. 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 6.

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

in which the call sequence is as described in the appropriate section of Chapter 8. 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 FSUNPCGSETMAXL(...)
CALL FSUNSPBCGSSETPRECTYPE(...)
CALL FSUNSPBCGSSETMAXL(...)
CALL FSUNSPFGMRSETGSTYPE(...)
CALL FSUNSPFGMRSETPRECTYPE(...)
CALL FSUNSPGMRSETGSTYPE(...)
CALL FSUNSPGMRSETPRECTYPE(...)
CALL FSUNSPTFQMRSETPRECTYPE(...)
CALL FSUNSPTFQMRSETMAXL(...)

```

where again the call sequences are described in the appropriate sections of Chapter 8.

5. SUNNONLINSOL module initialization

By default CVODE uses the SUNNONLINSOL implementation of Newton's method defined by the SUNNONLINSOL_NEWTON module (see §9.2). 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 9). For example, to create the SUNNONLINSOL_FIXEDPOINT solver call the function

```
CALL FSUNFIXEDPOINTINIT(...)
```

in which the call sequence is described in §9.3.

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)
Description	This function provides required problem and solution specifications, specifies optional inputs, allocates internal memory, and initializes CVODE.
Arguments	<p>T0 is the initial value of t.</p> <p>YO is an array of initial conditions.</p> <p>METH specifies the basic integration method: 1 for Adams (nonstiff) or 2 for BDF (stiff).</p> <p>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.</p> <p>RTOL is the relative tolerance (scalar).</p> <p>ATOL is the absolute tolerance (scalar or array).</p> <p>IOUT is an integer array of length 21 for integer optional outputs.</p> <p>ROUT is a real array of length 6 for real optional outputs.</p> <p>IPAR is an integer array of user data which will be passed unmodified to all user-provided routines.</p> <p>RPAR is a real array of user data which will be passed unmodified to all user-provided routines.</p>
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 5.2.

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 `FCVMALLOC`, 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 §5.5 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 $k = i - j + \text{MU} + 1$ ($k = 1 \cdots \text{ML} + \text{MU} + 1$) 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:

```
CALL FCVBANDSETJAC(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 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 `sunmatrix_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 `Y0` 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 FCVP SOL(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 **FCVP SOL** 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:

```
SUBROUTINE FCVP SET(T, Y, FY, JOK, JCUR, GAMMA, H, IPAR, RPAR, IER)
  DIMENSION Y(*), FY(*), EWT(*), IPAR(*), RPAR(*)
```

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. $\sqrt{\sum(\rho_i * EWT[i])^2} < \text{DELTA}$.
- (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 5.2.

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 9).

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 §4.5.6 and §B.2). The current values of the optional outputs are available in IOUT and ROUT (see Table 5.2).

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 ($0 \leq K \leq QU$). 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(T0, Y0, 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
```

Table 5.1: Keys for setting FCVODE optional inputs

Integer optional inputs (FCVSETIIN)		
Key	Optional input	Default value
MAX_ORD	Maximum LMM method order	5 (BDF), 12 (Adams)
MAX_NSTEPS	Maximum no. of internal steps before t_{out}	500
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

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

5.5 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 5.1), **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` (§4.5.7.1) 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 FCVMALLO. Table 5.2 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 §4.5.7 and §4.5.9.

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 5.2: Description of the FCVODE optional output arrays IOUT and ROUT

Integer output array IOUT		
Index	Optional output	CVODE function
CVODE main solver		
1	LENRW	CVodeGetWorkSpace
2	LENIW	CVodeGetWorkSpace
3	NST	CVodeGetNumSteps
4	NFE	CVodeGetNumRhsEvals
5	NETF	CVodeGetNumErrTestFails
6	NCFN	CVodeGetNumNonlinSolvConvFails
7	NNI	CVodeGetNumNonlinSolvIters
8	NSETUPS	CVodeGetNumLinSolvSetups
9	QU	CVodeGetLastOrder
10	QCUR	CVodeGetCurrentOrder
11	NOR	CVodeGetNumStabLimOrderReds
12	NGE	CVodeGetNumGEvals
CVLS linear 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
16	NFELS	CVDiagGetNumRhsEvals

Real output array ROUT		
Index	Optional output	CVODE function
1	HOU	CVodeGetActualInitStep
2	HU	CVodeGetLastStep
3	HCUR	CVodeGetCurrentStep
4	TCUR	CVodeGetCurrentTime
5	TOLSF	CVodeGetTolScaleFactor
6	UROUND	unit roundoff

This computes the EWT array normally defined by Eq. (2.7). 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.

5.6 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 CCodeRootInit.
- FCVROOTINFO interfaces to CCodeGetRootInfo.
- FCVROOTFREE interfaces to CCodeRootFree.

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) ($i = 1, \dots, NRTFN$) 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 decreasing, 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
```


5.7 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 §5.4 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, FSUNSPGMRINIT 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 §5.4 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 5.2. 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`.)

5.8 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
<code>FCVLOCFN</code>	<code>FCVgloc</code>	<code>CVLocalFn</code>
<code>FCVCOMMF</code>	<code>FCVcfn</code>	<code>CVCommFn</code>
<code>FCVJTIMES</code>	<code>FCVJtimes</code>	<code>CVLsJacTimesVecFn</code>
<code>FCVJTSETUP</code>	<code>FCVJTSetup</code>	<code>CVLsJacTimesSetupFn</code>

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 §5.3, 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 §5.4 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`, `FSUNSPGMRINIT`, `FSUNSPGMRINIT` or `FSUNSPTFQMRINIT`.

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 y for difference quotients (optional). A value of 0.0 indicates the default, $\sqrt{\text{unit roundoff}}$. IER is a return completion flag. A value of 0 indicates success, while a value of -1 indicates that a memory failure occurred or that an input had an illegal value.

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 §5.4 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 5.2. To obtain the optional outputs associated with the CVBBDPRE module, make the following call:

```
CALL FCVBBDOPT(LENRWBBD, LENIWBBBD, NGEBBBD)
```

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. LENIWBBBD is the length of integer preconditioner work space, in integer words. These sizes are local to the current processor. NGEBBBD 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, FSUNSPBCGSINIT, FSUNSPFGMRINIT, FSUNSPGMRINIT 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 §5.4.

Chapter 6

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 operations are described below and the implementations provided with SUNDIALS are described in the following sections.

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);  
    void (*nvsetarraypointer)(realtype *, N_Vector);  
    void (*nvlinearsum)(realtype, N_Vector, realtype, N_Vector, N_Vector);  
    void (*nvconst)(realtype, N_Vector);  
    void (*nvprod)(N_Vector, N_Vector, N_Vector);  
    void (*nvdiv)(N_Vector, N_Vector, N_Vector);  
    void (*nvscale)(realtype, N_Vector, N_Vector);  
    void (*nvabs)(N_Vector, N_Vector);  
    void (*nvinv)(N_Vector, N_Vector);  
    void (*nvaddconst)(N_Vector, realtype, N_Vector);  
    realtype (*nvdotprod)(N_Vector, N_Vector);  
    realtype (*nvmaxnorm)(N_Vector);  
    realtype (*nvwrmsnorm)(N_Vector, N_Vector);
```

```

realtype    (*nvwrmsnormmask)(N_Vector, N_Vector, N_Vector);
realtype    (*nvmin)(N_Vector);
realtype    (*nvwl2norm)(N_Vector, N_Vector);
realtype    (*nvlinorm)(N_Vector);
void        (*nvcompare)(realtype, N_Vector, N_Vector);
boolean_t   (*nvinvtest)(N_Vector, N_Vector);
boolean_t   (*nvconstrmask)(N_Vector, N_Vector, N_Vector);
realtype    (*nvminquotient)(N_Vector, N_Vector);
int         (*nvlinearcombination)(int, realtype*, N_Vector*, N_Vector);
int         (*nvscaleaddmulti)(int, realtype*, N_Vector, N_Vector*, N_Vector*);
int         (*nvdotprodmulti)(int, N_Vector, N_Vector*, realtype*);
int         (*nvlinearsumvectorarray)(int, realtype, N_Vector*, realtype,
                                     N_Vector*, N_Vector*);
int         (*nvscalevectorarray)(int, realtype*, N_Vector*, N_Vector*);
int         (*nvconstvectorarray)(int, realtype, N_Vector*);
int         (*nvwrmsnomrvectorarray)(int, N_Vector*, N_Vector*, realtype*);
int         (*nvwrmsnomrmaskvectorarray)(int, N_Vector*, N_Vector*, N_Vector,
                                     realtype*);
int         (*nvscaleaddmultivectorarray)(int, int, realtype*, N_Vector*,
                                     N_Vector**, N_Vector**);
int         (*nvlinearcombinationvectorarray)(int, int, realtype*, N_Vector**,
                                     N_Vector*);
};

```

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);
}

```

Table 6.2 contains a complete list of all standard vector operations defined by the generic NVECTOR module. Tables 6.3 and 6.4 list *optional* fused and vector array operations respectively. These operations are intended to increase data reuse, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. If a particular NVECTOR implementation defines one of the fused or vector array operations as `NULL`, the NVECTOR interface will call one of the standard vector operations as necessary.

Finally, note that the generic NVECTOR module defines the functions `N_VCloneVectorArray` and `N_VCloneVectorArrayEmpty`. Both functions create (by cloning) an array of `count` variables of type `N_Vector`, each of the same type as an existing `N_Vector`. Their prototypes are

```

N_Vector *N_VCloneVectorArray(int count, N_Vector w);
N_Vector *N_VCloneVectorArrayEmpty(int count, N_Vector w);

```

and their definitions are based on the implementation-specific `N_VClone` and `N_VCloneEmpty` operations, respectively.

An array of variables of type `N_Vector` can be destroyed by calling `N_VDestroyVectorArray`, whose prototype is

```

void N_VDestroyVectorArray(N_Vector *vs, int count);

```

and whose definition is based on the implementation-specific `N_VDestroy` operation.

A particular implementation of the NVECTOR module must:

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

Vector ID	Vector type	ID Value
SUNDIALS_NVEC_SERIAL	Serial	0
SUNDIALS_NVEC_PARALLEL	Distributed memory parallel (MPI)	1
SUNDIALS_NVEC_OPENMP	OpenMP shared memory parallel	2
SUNDIALS_NVEC_PTHREADS	PThreads shared memory parallel	3
SUNDIALS_NVEC_PARHYP	<i>hypre</i> ParHyp parallel vector	4
SUNDIALS_NVEC_PETSC	PETSc parallel vector	5
SUNDIALS_NVEC_CUSTOM	User-provided custom vector	6

- 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**.

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

Table 6.2: Description of the NVECTOR operations

Name	Usage and Description
<code>N_VGetVectorID</code>	<pre>id = N_VGetVectorID(w);</pre> <p>Returns the vector type identifier for the vector <code>w</code>. It is used to determine the vector implementation type (e.g. serial, parallel, ...) from the abstract <code>N_Vector</code> interface. Returned values are given in Table 6.1.</p>
<code>N_VClone</code>	<pre>v = N_VClone(w);</pre> <p>Creates a new <code>N_Vector</code> of the same type as an existing vector <code>w</code> and sets the <code>ops</code> field. It does not copy the vector, but rather allocates storage for the new vector.</p>
<code>N_VCloneEmpty</code>	<pre>v = N_VCloneEmpty(w);</pre> <p>Creates a new <code>N_Vector</code> of the same type as an existing vector <code>w</code> and sets the <code>ops</code> field. It does not allocate storage for data.</p>
<code>N_VDestroy</code>	<pre>N_VDestroy(v);</pre> <p>Destroys the <code>N_Vector</code> <code>v</code> and frees memory allocated for its internal data.</p>
<code>N_VSpace</code>	<pre>N_VSpace(nvSpec, &lrw, &liw);</pre> <p>Returns storage requirements for one <code>N_Vector</code>. <code>lrw</code> contains the number of realtype words and <code>liw</code> 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.</p>
<code>N_VGetArrayPointer</code>	<pre>vdata = N_VGetArrayPointer(v);</pre> <p>Returns a pointer to a <code>realtype</code> array from the <code>N_Vector</code> <code>v</code>. Note that this assumes that the internal data in <code>N_Vector</code> is a contiguous array of <code>realtype</code>. 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.</p>
<code>N_VSetArrayPointer</code>	<pre>N_VSetArrayPointer(vdata, v);</pre> <p>Overwrites the data in an <code>N_Vector</code> with a given array of <code>realtype</code>. Note that this assumes that the internal data in <code>N_Vector</code> is a contiguous array of <code>realtype</code>. 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.</p>
<i>continued on next page</i>	

<i>continued from last page</i>	
Name	Usage and Description
N_VLinearSum	N_VLinearSum(a, x, b, y, z); Performs the operation $z = ax + by$, where a and b are realtype scalars and x and y are of type N_Vector : $z_i = ax_i + by_i$, $i = 0, \dots, n-1$.
N_VConst	N_VConst(c, z); Sets all components of the N_Vector z to realtype c : $z_i = c$, $i = 0, \dots, n-1$.
N_VProd	N_VProd(x, y, z); Sets the N_Vector z to be the component-wise product of the N_Vector inputs x and y : $z_i = x_i y_i$, $i = 0, \dots, n-1$.
N_VDiv	N_VDiv(x, y, z); 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.
N_VScale	N_VScale(c, x, z); Scales the N_Vector x by the realtype scalar c and returns the result in z : $z_i = cx_i$, $i = 0, \dots, n-1$.
N_VAbs	N_VAbs(x, z); Sets the components of the N_Vector z to be the absolute values of the components of the N_Vector x : $y_i = x_i $, $i = 0, \dots, n-1$.
N_VInv	N_VInv(x, z); Sets the components of the N_Vector z to be the inverses of the components of the N_Vector x : $z_i = 1.0/x_i$, $i = 0, \dots, n-1$. This routine may not check for division by 0. It should be called only with an x which is guaranteed to have all nonzero components.
N_VAddConst	N_VAddConst(x, b, z); Adds the realtype scalar b to all components of x and returns the result in the N_Vector z : $z_i = x_i + b$, $i = 0, \dots, n-1$.
N_VDotProd	d = N_VDotProd(x, y); Returns the value of the ordinary dot product of x and y : $d = \sum_{i=0}^{n-1} x_i y_i$.
N_VMaxNorm	m = N_VMaxNorm(x); Returns the maximum norm of the N_Vector x : $m = \max_i x_i $.
<i>continued on next page</i>	

<i>continued from last page</i>	
Name	Usage and Description
N_VWrmsNorm	$\mathbf{m} = \text{N_VWrmsNorm}(\mathbf{x}, \mathbf{w})$ Returns the weighted root-mean-square norm of the N_Vector \mathbf{x} with realtype weight vector \mathbf{w} : $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i)^2\right) / n}$.
N_VWrmsNormMask	$\mathbf{m} = \text{N_VWrmsNormMask}(\mathbf{x}, \mathbf{w}, \text{id});$ Returns the weighted root mean square norm of the N_Vector \mathbf{x} with realtype weight vector \mathbf{w} built using only the elements of \mathbf{x} corresponding to positive elements of the N_Vector id : $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i H(\text{id}_i))^2\right) / n}, \text{ where } H(\alpha) = \begin{cases} 1 & \alpha > 0 \\ 0 & \alpha \leq 0 \end{cases}$
N_VMin	$\mathbf{m} = \text{N_VMin}(\mathbf{x});$ Returns the smallest element of the N_Vector \mathbf{x} : $m = \min_i x_i$.
N_VWL2Norm	$\mathbf{m} = \text{N_VWL2Norm}(\mathbf{x}, \mathbf{w});$ Returns the weighted Euclidean ℓ_2 norm of the N_Vector \mathbf{x} with realtype weight vector \mathbf{w} : $m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2}$.
N_VL1Norm	$\mathbf{m} = \text{N_VL1Norm}(\mathbf{x});$ Returns the ℓ_1 norm of the N_Vector \mathbf{x} : $m = \sum_{i=0}^{n-1} x_i $.
N_VCompare	$\text{N_VCompare}(\mathbf{c}, \mathbf{x}, \mathbf{z});$ Compares the components of the N_Vector \mathbf{x} to the realtype scalar \mathbf{c} and returns an N_Vector \mathbf{z} such that: $z_i = 1.0$ if $ x_i \geq c$ and $z_i = 0.0$ otherwise.
N_VInvTest	$\mathbf{t} = \text{N_VInvTest}(\mathbf{x}, \mathbf{z});$ Sets the components of the N_Vector \mathbf{z} to be the inverses of the components of the N_Vector \mathbf{x} , with prior testing for zero values: $z_i = 1.0/x_i, i = 0, \dots, n-1$. This routine returns a boolean assigned to SUNTRUE if all components of \mathbf{x} are nonzero (successful inversion) and returns SUNFALSE otherwise.
N_VConstrMask	$\mathbf{t} = \text{N_VConstrMask}(\mathbf{c}, \mathbf{x}, \mathbf{m});$ 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.
<i>continued on next page</i>	

<i>continued from last page</i>	
Name	Usage and Description
N_VMinQuotient	<pre>minq = N_VMinQuotient(num, denom);</pre> <p>This routine returns the minimum of the quotients obtained by term-wise dividing 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.</p>

Table 6.3: Description of the NVECTOR fused operations

Name	Usage and Description
N_VLinearCombination	<pre>ier = N_VLinearCombination(nv, c, X, z);</pre> <p>This routine computes the linear combination of n_v vectors with n elements:</p> $z_i = \sum_{j=0}^{n_v-1} c_j x_{j,i}, \quad i = 0, \dots, n-1,$ <p>where c is an array of n_v scalars (type realtype*), X is an array of n_v vectors (type N_Vector*), and z is the output vector (type N_Vector). If the output vector z is one of the vectors in X, then it <i>must</i> be the first vector in the vector array. The operation returns 0 for success and a non-zero value otherwise.</p>
N_VScaleAddMulti	<pre>ier = N_VScaleAddMulti(nv, c, x, Y, Z);</pre> <p>This routine scales and adds one vector to n_v vectors with n elements:</p> $z_{j,i} = c_j x_i + y_{j,i}, \quad j = 0, \dots, n_v-1 \quad i = 0, \dots, n-1,$ <p>where c is an array of n_v scalars (type realtype*), x is the vector (type N_Vector) to be scaled and added to each vector in the vector array of n_v vectors Y (type N_Vector*), and Z (type N_Vector*) is a vector array of n_v output vectors. The operation returns 0 for success and a non-zero value otherwise.</p>
<i>continued on next page</i>	

<i>continued from last page</i>	
Name	Usage and Description
N_VDotProdMulti	<p><code>ier = N_VDotProdMulti(nv, x, Y, d);</code> This routine computes the dot product of a vector with n_v other vectors:</p> $d_j = \sum_{i=0}^{n-1} x_i y_{j,i}, \quad j = 0, \dots, n_v - 1,$ <p>where d (type <code>realtype*</code>) is an array of n_v scalars containing the dot products of the vector x (type <code>N_Vector</code>) with each of the n_v vectors in the vector array Y (type <code>N_Vector*</code>). The operation returns 0 for success and a non-zero value otherwise.</p>

Table 6.4: Description of the NVECTOR vector array operations

Name	Usage and Description
N_VLinearSumVectorArray	<p><code>ier = N_VLinearSumVectorArray(nv, a, X, b, Y, Z);</code> This routine computes the linear sum of two vector arrays containing n_v vectors of n elements:</p> $z_{j,i} = ax_{j,i} + by_{j,i}, \quad i = 0, \dots, n-1 \quad j = 0, \dots, n_v - 1,$ <p>where a and b are <code>realtype</code> scalars and X, Y, and Z are arrays of n_v vectors (type <code>N_Vector*</code>). The operation returns 0 for success and a non-zero value otherwise.</p>
N_VScaleVectorArray	<p><code>ier = N_VScaleVectorArray(nv, c, X, Z);</code> This routine scales each vector of n elements in a vector array of n_v vectors by a potentially different constant:</p> $z_{j,i} = c_j x_{j,i}, \quad i = 0, \dots, n-1 \quad j = 0, \dots, n_v - 1,$ <p>where c is an array of n_v scalars (type <code>realtype*</code>) and X and Z are arrays of n_v vectors (type <code>N_Vector*</code>). The operation returns 0 for success and a non-zero value otherwise.</p>
<i>continued on next page</i>	

<i>continued from last page</i>	
Name	Usage and Description
N_VConstVectorArray	<p>ier = N_VConstVectorArray(nv, c, X);</p> <p>This routine sets each element in a vector of n elements in a vector array of n_v vectors to the same value:</p> $z_{j,i} = c, \quad i = 0, \dots, n-1 \quad j = 0, \dots, n_v-1,$ <p>where c is a realtype scalar and X is an array of n_v vectors (type N_Vector*). The operation returns 0 for success and a non-zero value otherwise.</p>
N_VWrmsNormVectorArray	<p>ier = N_VWrmsNormVectorArray(nv, X, W, m);</p> <p>This routine computes the weighted root mean square norm of n_v vectors with n elements:</p> $m_j = \left(\frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i} w_{j,i})^2 \right)^{1/2}, \quad j = 0, \dots, n_v-1,$ <p>where m (type realtype*) contains the n_v norms of the vectors in the vector array X (type N_Vector*) with corresponding weight vectors W (type N_Vector*). The operation returns 0 for success and a non-zero value otherwise.</p>
N_VWrmsNormMaskVectorArray	<p>ier = N_VWrmsNormMaskVectorArray(nv, X, W, id, m);</p> <p>This routine computes the masked weighted root mean square norm of n_v vectors with n elements:</p> $m_j = \left(\frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i} w_{j,i} H(id_i))^2 \right)^{1/2}, \quad j = 0, \dots, n_v-1,$ <p>$H(id_i) = 1$ for $id_i > 0$ and is zero otherwise, m (type realtype*) contains the n_v norms of the vectors in the vector array X (type N_Vector*) with corresponding weight vectors W (type N_Vector*) and mask vector id (type N_Vector). The operation returns 0 for success and a non-zero value otherwise.</p>
<i>continued on next page</i>	

continued from last page	
Name	Usage and Description
N_VScaleAddMultiVectorArray	<p><code>ier = N_VScaleAddMultiVectorArray(nv, ns, c, X, YY, ZZ);</code></p> <p>This routine scales and adds a vector in a vector array of n_v vectors to the corresponding vector in n_s vector arrays:</p> $z_{j,i} = \sum_{k=0}^{n_s-1} c_k x_{k,j,i}, \quad i = 0, \dots, n-1 \quad j = 0, \dots, n_v-1,$ <p>where c is an array of n_s scalars (type <code>realtype*</code>), X is a vector array of n_v vectors (type <code>idN_Vector*</code>) to be scaled and added to the corresponding vector in each of the n_s vector arrays in the array of vector arrays YY (type <code>N_Vector**</code>) and stored in the output array of vector arrays ZZ (type <code>N_Vector**</code>). The operation returns 0 for success and a non-zero value otherwise.</p>
N_VLinearCombinationVectorArray	<p><code>ier = N_VLinearCombinationVectorArray(nv, ns, c, XX, Z);</code></p> <p>This routine computes the linear combination of n_s vector arrays containing n_v vectors with n elements:</p> $z_{j,i} = \sum_{k=0}^{n_s-1} c_k x_{k,j,i}, \quad i = 0, \dots, n-1 \quad j = 0, \dots, n_v-1,$ <p>where c is an array of n_s scalars (type <code>realtype*</code>), XX (type <code>N_Vector**</code>) 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 <code>N_Vector*</code>). If the output vector array Z is one of the vector arrays in XX, then it <i>must</i> be the first vector array in XX. The operation returns 0 for success and a non-zero value otherwise.</p>

6.1 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.

The following macros are provided to access the content of an NVECTOR_SERIAL vector. The suffix *_S* in the names denotes the serial version.

- NV_CONTENT_S

This routine gives access to the contents of the serial vector `N_Vector`.

The assignment `v_cont = NV_CONTENT_S(v)` sets `v_cont` to be a pointer to the serial `N_Vector` content structure.

Implementation:

```
#define NV_CONTENT_S(v) ( (N_VectorContent_Serial)(v->content) )
```

- `NV_OWN_DATA_S`, `NV_DATA_S`, `NV_LENGTH_S`

These macros give individual access to the parts of the content of a serial `N_Vector`.

The assignment `v_data = NV_DATA_S(v)` sets `v_data` to be a pointer to the first component of the data for the `N_Vector` `v`. The assignment `NV_DATA_S(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

The assignment `v_len = NV_LENGTH_S(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_S(v) = len_v` sets the length of `v` to be `len_v`.

Implementation:

```
#define NV_OWN_DATA_S(v) ( NV_CONTENT_S(v)->own_data )
```

```
#define NV_DATA_S(v) ( NV_CONTENT_S(v)->data )
```

```
#define NV_LENGTH_S(v) ( NV_CONTENT_S(v)->length )
```

- `NV_Ith_S`

This macro gives access to the individual components of the data array of an `N_Vector`.

The assignment `r = NV_Ith_S(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_S(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

Here `i` ranges from 0 to $n - 1$ for a vector of length n .

Implementation:

```
#define NV_Ith_S(v,i) ( NV_DATA_S(v)[i] )
```

The `NVECTOR_SERIAL` module defines serial implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4. Their names are obtained from those in Tables 6.2, 6.3, and 6.4. by appending the suffix `_Serial` (e.g. `N_VDestroy_Serial`). The module `NVECTOR_SERIAL` provides the following additional user-callable routines:

- `N_VNew_Serial`

This function creates and allocates memory for a serial `N_Vector`. Its only argument is the vector length.

```
N_Vector N_VNew_Serial(sunindextype vec_length);
```

- `N_VNewEmpty_Serial`

This function creates a new serial `N_Vector` with an empty (`NULL`) data array.

```
N_Vector N_VNewEmpty_Serial(sunindextype vec_length);
```

- `N_VMake_Serial`

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.)

```
N_Vector N_VMake_Serial(sunindextype vec_length, realtype *v_data);
```

- `N_VCloneVectorArray_Serial`

This function creates (by cloning) an array of `count` serial vectors.

```
N_Vector *N_VCloneVectorArray_Serial(int count, N_Vector w);
```

- `N_VCloneVectorArrayEmpty_Serial`

This function creates (by cloning) an array of `count` serial vectors, each with an empty (NULL) data array.

```
N_Vector *N_VCloneVectorArrayEmpty_Serial(int count, N_Vector w);
```

- `N_VDestroyVectorArray_Serial`

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`.

```
void N_VDestroyVectorArray_Serial(N_Vector *vs, int count);
```

- `N_VGetLength_Serial`

This function returns the number of vector elements.

```
sunindextype N_VGetLength_Serial(N_Vector v);
```

- `N_VPrint_Serial`

This function prints the content of a serial vector to `stdout`.

```
void N_VPrint_Serial(N_Vector v);
```

- `N_VPrintFile_Serial`

This function prints the content of a serial vector to `outfile`.

```
void N_VPrintFile_Serial(N_Vector v, FILE *outfile);
```

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.

For solvers that include a Fortran 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.

6.2 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.

The following macros are provided to access the content of a NVECTOR_PARALLEL vector. The suffix `_P` in the names denotes the distributed memory parallel version.

- **NV_CONTENT_P**

This macro gives access to the contents of the parallel vector `N_Vector`.

The assignment `v_cont = NV_CONTENT_P(v)` sets `v_cont` to be a pointer to the `N_Vector` content structure of type `struct _N_VectorContent_Parallel`.

Implementation:

```
#define NV_CONTENT_P(v) ( (_N_VectorContent_Parallel)(v->content) )
```

- **NV_OWN_DATA_P, NV_DATA_P, NV_LOCLENGTH_P, NV_GLOBLENGTH_P**

These macros give individual access to the parts of the content of a parallel `N_Vector`.

The assignment `v_data = NV_DATA_P(v)` sets `v_data` to be a pointer to the first component of the local data for the `N_Vector` `v`. The assignment `NV_DATA_P(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

The assignment `v_llen = NV_LOCLENGTH_P(v)` sets `v_llen` to be the length of the local part of `v`. The call `NV_LENGTH_P(v) = llen_v` sets the local length of `v` to be `llen_v`.

The assignment `v_glen = NV_GLOBLENGTH_P(v)` sets `v_glen` to be the global length of the vector `v`. The call `NV_GLOBLENGTH_P(v) = glen_v` sets the global length of `v` to be `glen_v`.

Implementation:

```

#define NV_OWN_DATA_P(v)    ( NV_CONTENT_P(v)->own_data )
#define NV_DATA_P(v)        ( NV_CONTENT_P(v)->data )
#define NV_LOCLENGTH_P(v)   ( NV_CONTENT_P(v)->local_length )
#define NV_GLOBLENGTH_P(v) ( NV_CONTENT_P(v)->global_length )

```

- **NV_COMM_P**

This macro provides access to the MPI communicator used by the NVECTOR_PARALLEL vectors.

Implementation:

```
#define NV_COMM_P(v) ( NV_CONTENT_P(v)->comm )
```

- **NV_Ith_P**

This macro gives access to the individual components of the local data array of an `N_Vector`.

The assignment `r = NV_Ith_P(v,i)` sets `r` to be the value of the `i`-th component of the local part of `v`. The assignment `NV_Ith_P(v,i) = r` sets the value of the `i`-th component of the local part of `v` to be `r`.

Here `i` ranges from 0 to `n - 1`, where `n` is the local length.

Implementation:

```
#define NV_Ith_P(v,i) ( NV_DATA_P(v)[i] )
```

The NVECTOR_PARALLEL module defines parallel implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4. Their names are obtained from those in Tables 6.2, 6.3, and 6.4 by appending the suffix `_Parallel` (e.g. `N_VDestroy_Parallel`). The module NVECTOR_PARALLEL provides the following additional user-callable routines:

- `N_VNew_Parallel`

This function creates and allocates memory for a parallel vector.

```
N_Vector N_VNew_Parallel(MPI_Comm comm,
                        sunindextype local_length,
                        sunindextype global_length);
```

- `N_VNewEmpty_Parallel`

This function creates a new parallel `N_Vector` with an empty (NULL) data array.

```
N_Vector N_VNewEmpty_Parallel(MPI_Comm comm,
                              sunindextype local_length,
                              sunindextype global_length);
```

- `N_VMake_Parallel`

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.)

```
N_Vector N_VMake_Parallel(MPI_Comm comm,
                          sunindextype local_length,
                          sunindextype global_length,
                          realtype *v_data);
```

- `N_VCloneVectorArray_Parallel`

This function creates (by cloning) an array of `count` parallel vectors.

```
N_Vector *N_VCloneVectorArray_Parallel(int count, N_Vector w);
```

- `N_VCloneVectorArrayEmpty_Parallel`

This function creates (by cloning) an array of `count` parallel vectors, each with an empty (NULL) data array.

```
N_Vector *N_VCloneVectorArrayEmpty_Parallel(int count, N_Vector w);
```

- `N_VDestroyVectorArray_Parallel`

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`.

```
void N_VDestroyVectorArray_Parallel(N_Vector *vs, int count);
```

- `N_VGetLength_Parallel`

This function returns the number of vector elements (global vector length).

```
sunindextype N_VGetLength_Parallel(N_Vector v);
```

- `N_VGetLocalLength_Parallel`

This function returns the local vector length.

```
sunindextype N_VGetLocalLength_Parallel(N_Vector v);
```

- `N_VPrint_Parallel`

This function prints the local content of a parallel vector to `stdout`.

```
void N_VPrint_Parallel(N_Vector v);
```

- `N_VPrintFile_Parallel`

This function prints the local content of a parallel vector to `outfile`.

```
void N_VPrintFile_Parallel(N_Vector v, FILE *outfile);
```

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.



For solvers that include a Fortran 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.



6.3 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 following macros are provided to access the content of an NVECTOR_OPENMP vector. The suffix `_OMP` in the names denotes the OpenMP version.

- **NV_CONTENT_OMP**

This routine gives access to the contents of the OpenMP vector `N_Vector`.

The assignment `v_cont = NV_CONTENT_OMP(v)` sets `v_cont` to be a pointer to the OpenMP `N_Vector` content structure.

Implementation:

```
#define NV_CONTENT_OMP(v) ( (N_VectorContent_OpenMP)(v->content) )
```

- **NV_OWN_DATA_OMP, NV_DATA_OMP, NV_LENGTH_OMP, NV_NUM_THREADS_OMP**

These macros give individual access to the parts of the content of a OpenMP `N_Vector`.

The assignment `v_data = NV_DATA_OMP(v)` sets `v_data` to be a pointer to the first component of the data for the `N_Vector` `v`. The assignment `NV_DATA_OMP(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

The assignment `v_len = NV_LENGTH_OMP(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_OMP(v) = len_v` sets the length of `v` to be `len_v`.

The assignment `v_num_threads = NV_NUM_THREADS_OMP(v)` sets `v_num_threads` to be the number of threads from `v`. On the other hand, the call `NV_NUM_THREADS_OMP(v) = num_threads_v` sets the number of threads for `v` to be `num_threads_v`.

Implementation:

```
#define NV_OWN_DATA_OMP(v) ( NV_CONTENT_OMP(v)->own_data )
```

```
#define NV_DATA_OMP(v) ( NV_CONTENT_OMP(v)->data )
```

```
#define NV_LENGTH_OMP(v) ( NV_CONTENT_OMP(v)->length )
```

```
#define NV_NUM_THREADS_OMP(v) ( NV_CONTENT_OMP(v)->num_threads )
```

- **NV_Ith_OMP**

This macro gives access to the individual components of the data array of an `N_Vector`.

The assignment `r = NV_Ith_OMP(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_OMP(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

Here `i` ranges from 0 to $n - 1$ for a vector of length `n`.

Implementation:

```
#define NV_Ith_OMP(v,i) ( NV_DATA_OMP(v)[i] )
```

The NVECTOR_OPENMP module defines OpenMP implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4. Their names are obtained from those in Tables 6.2, 6.3, and 6.4 by appending the suffix `_OpenMP` (e.g. `N_VDestroy_OpenMP`). The module NVECTOR_OPENMP provides the following additional user-callable routines:

- **N_VNew_OpenMP**

This function creates and allocates memory for a OpenMP `N_Vector`. Arguments are the vector length and number of threads.

```
N_Vector N_VNew_OpenMP(sunindextype vec_length, int num_threads);
```

- **N_VNewEmpty_OpenMP**

This function creates a new OpenMP `N_Vector` with an empty (NULL) data array.

```
N_Vector N_VNewEmpty_OpenMP(sunindextype vec_length, int num_threads);
```

- `N_VMake_OpenMP`

This function creates and allocates memory for an OpenMP vector with user-provided data array. (This function does *not* allocate memory for `v_data` itself.)

```
N_Vector N_VMake_OpenMP(sunindextype vec_length, realtype *v_data,
                        int num_threads);
```

- `N_VCloneVectorArray_OpenMP`

This function creates (by cloning) an array of `count` OpenMP vectors.

```
N_Vector *N_VCloneVectorArray_OpenMP(int count, N_Vector w);
```

- `N_VCloneVectorArrayEmpty_OpenMP`

This function creates (by cloning) an array of `count` OpenMP vectors, each with an empty (NULL) data array.

```
N_Vector *N_VCloneVectorArrayEmpty_OpenMP(int count, N_Vector w);
```

- `N_VDestroyVectorArray_OpenMP`

This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_OpenMP` or with `N_VCloneVectorArrayEmpty_OpenMP`.

```
void N_VDestroyVectorArray_OpenMP(N_Vector *vs, int count);
```

- `N_VGetLength_OpenMP`

This function returns number of vector elements.

```
sunindextype N_VGetLength_OpenMP(N_Vector v);
```

- `N_VPrint_OpenMP`

This function prints the content of an OpenMP vector to `stdout`.



```
void N_VPrint_OpenMP(N_Vector v);
```

- `N_VPrintFile_OpenMP`

This function prints the content of an OpenMP vector to `outfile`.

```
void N_VPrintFile_OpenMP(N_Vector v, FILE *outfile);
```

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. 

For solvers that include a Fortran 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.

6.4 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.

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] )
```

The NVECTOR_PTHREADS module defines Pthreads implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4. Their names are obtained from those in Tables 6.2, 6.3, and 6.4 by appending the suffix `_Pthreads` (e.g. `N_VDestroy_Pthreads`). The module NVECTOR_PTHREADS provides the following additional user-callable routines:

- **N_VNew_Pthreads**

This function creates and allocates memory for a Pthreads `N_Vector`. Arguments are the vector length and number of threads.

```
N_Vector N_VNew_Pthreads(sunindextype vec_length, int num_threads);
```

- **N_VNewEmpty_Pthreads**

This function creates a new Pthreads `N_Vector` with an empty (NULL) data array.

```
N_Vector N_VNewEmpty_Pthreads(sunindextype vec_length, int num_threads);
```

- **N_VMake_Pthreads**

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.)

```
N_Vector N_VMake_Pthreads(sunindextype vec_length, realtype *v_data,
                          int num_threads);
```

- **N_VCloneVectorArray_Pthreads**

This function creates (by cloning) an array of `count` Pthreads vectors.

```
N_Vector *N_VCloneVectorArray_Pthreads(int count, N_Vector w);
```

- **N_VCloneVectorArrayEmpty_Pthreads**

This function creates (by cloning) an array of `count` Pthreads vectors, each with an empty (NULL) data array.

```
N_Vector *N_VCloneVectorArrayEmpty_Pthreads(int count, N_Vector w);
```

- **N_VDestroyVectorArray_Pthreads**

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`.

```
void N_VDestroyVectorArray_Pthreads(N_Vector *vs, int count);
```

- **N_VGetLength_Pthreads**

This function returns the number of vector elements.

```
sunindextype N_VGetLength_Pthreads(N_Vector v);
```

- **N_VPrint_Pthreads**

This function prints the content of a Pthreads vector to `stdout`.

```
void N_VPrint_Pthreads(N_Vector v);
```

- **N_VPrintFile_Pthreads**

This function prints the content of a Pthreads vector to `outfile`.

```
void N_VPrintFile_Pthreads(N_Vector v, FILE *outfile);
```

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.

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.

6.5 The NVECTOR_PARHYP implementation

The `NVECTOR_PARHYP` implementation of the `NVECTOR` module provided with `SUNDIALS` is a wrapper around `hypr`'s `ParVector` class. Most of the vector kernels simply call `hypr` 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 `hypr_ParVector`, an MPI communicator, and a boolean flag `own_parvector` indicating ownership of the `hypr` parallel vector object `x`.

```
struct _N_VectorContent_ParHyp {
    sunindextype local_length;
    sunindextype global_length;
    booleantype own_parvector;
    MPI_Comm comm;
    hypr_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.

The `NVECTOR_PARHYP` module defines implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4, except for `N_VSetArrayPointer` and `N_VGetArrayPointer`, because accessing raw vector data is handled by low-level `hypr` 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 `hypr` vector first, and then use `hypr` methods to access the data. Usage examples of `NVECTOR_PARHYP` are provided in the `cvAdvDiff_non_ph.c` example program for `CVODE` [27] and the `ark_diurnal_kry_ph.c` example program for `ARKODE` [34].

The names of `parhyp` methods are obtained from those in Tables 6.2, 6.3, and 6.4 by appending the suffix `_ParHyp` (e.g. `N_VDestroy_ParHyp`). The module `NVECTOR_PARHYP` provides the following additional user-callable routines:

- `N_VNewEmpty_ParHyp`

This function creates a new `parhyp` `N_Vector` with the pointer to the `hypr` vector set to `NULL`.


```
N_Vector N_VNewEmpty_ParHyp(MPI_Comm comm,
                             sunindextype local_length,
                             sunindextype global_length);
```

- **N_VMake_ParHyp**

This function creates an `N_Vector` wrapper around an existing *hypre* parallel vector. It does *not* allocate memory for `x` itself.

```
N_Vector N_VMake_ParHyp(hypre_ParVector *x);
```

- **N_VGetVector_ParHyp**

This function returns a pointer to the underlying *hypre* vector.

```
hypre_ParVector *N_VGetVector_ParHyp(N_Vector v);
```

- **N_VCloneVectorArray_ParHyp**

This function creates (by cloning) an array of `count` parallel vectors.

```
N_Vector *N_VCloneVectorArray_ParHyp(int count, N_Vector w);
```

- **N_VCloneVectorArrayEmpty_ParHyp**

This function creates (by cloning) an array of `count` parallel vectors, each with an empty (NULL) data array.

```
N_Vector *N_VCloneVectorArrayEmpty_ParHyp(int count, N_Vector w);
```

- **N_VDestroyVectorArray_ParHyp**

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`.

```
void N_VDestroyVectorArray_ParHyp(N_Vector *vs, int count);
```

- **N_VPrint_ParHyp**

This function prints the local content of a parhyp vector to `stdout`.

```
void N_VPrint_ParHyp(N_Vector v);
```

- **N_VPrintFile_ParHyp**

This function prints the local content of a parhyp vector to `outfile`.

```
void N_VPrintFile_ParHyp(N_Vector v, FILE *outfile);
```

Notes

- When there is a need to access components of an `N_Vector_ParHyp`, `v`, it is recommended to extract the *hypre* vector via `x_vec = N_VGetVector_ParHyp(v)` and then access components using appropriate *hypre* functions.
- `N_VNewEmpty_ParHyp`, `N_VMake_ParHyp`, and `N_VCloneVectorArrayEmpty_ParHyp` set the field *own_parvector* to `SUNFALSE`. `N_VDestroy_ParHyp` and `N_VDestroyVectorArray_ParHyp` will not attempt to delete an underlying *hypre* vector for any `N_Vector` with *own_parvector* set to `SUNFALSE`. In such a case, it is the user's responsibility to delete the underlying vector.
- To maximize efficiency, vector operations in the `NVECTOR_PARHYP` implementation that have more than one `N_Vector` argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.



6.6 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.

The NVECTOR_PETSC module defines implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4, 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 [26].

The names of vector operations are obtained from those in Tables 6.2, 6.3, and 6.4 by appending the suffix `_Petsc` (e.g. `N_VDestroy_Petsc`). The module NVECTOR_PETSC provides the following additional user-callable routines:

- `N_VNewEmpty_Petsc`

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_Vector N_VNewEmpty_Petsc(MPI_Comm comm,
                           sunindextype local_length,
                           sunindextype global_length);
```

- `N_VMake_Petsc`

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_Vector N_VMake_Petsc(Vec *pvec);
```

- `N_VGetVector_Petsc`

This function returns a pointer to the underlying PETSc vector.

```
Vec *N_VGetVector_Petsc(N_Vector v);
```

- `N_VCloneVectorArray_Petsc`

This function creates (by cloning) an array of `count` NVECTOR_PETSC vectors.

```
N_Vector *N_VCloneVectorArray_Petsc(int count, N_Vector w);
```

- `N_VCloneVectorArrayEmpty_Petsc`

This function creates (by cloning) an array of `count` `NVECTOR_PETSC` vectors, each with pointers to PETSc vectors set to `(NULL)`.

```
N_Vector *N_VCloneVectorArrayEmpty_Petsc(int count, N_Vector w);
```

- `N_VDestroyVectorArray_Petsc`

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`.

```
void N_VDestroyVectorArray_Petsc(N_Vector *vs, int count);
```

- `N_VPrint_Petsc`

This function prints the global content of a wrapped PETSc vector to `stdout`.

```
void N_VPrint_Petsc(N_Vector v);
```

- `N_VPrintFile_Petsc`

This function prints the global content of a wrapped PETSc vector to `fname`.

```
void N_VPrintFile_Petsc(N_Vector v, const char fname[]);
```

Notes

- When there is a need to access components of an `N_Vector_Petsc`, `v`, it is recommended to extract the PETSc vector via `x_vec = N_VGetVector_Petsc(v)` and then access components using appropriate PETSc functions.
- The functions `N_VNewEmpty_Petsc`, `N_VMake_Petsc`, and `N_VCloneVectorArrayEmpty_Petsc` set the field `own_data` to `SUNFALSE`. `N_VDestroy_Petsc` and `N_VDestroyVectorArray_Petsc` will not attempt to free the pointer `pvec` for any `N_Vector` with `own_data` set to `SUNFALSE`. In such a case, it is the user's responsibility to deallocate the `pvec` pointer.
- To maximize efficiency, vector operations in the `NVECTOR_PETSC` implementation that have more than one `N_Vector` argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.



6.7 The NVECTOR_CUDA implementation

The `NVECTOR_CUDA` module is an experimental `NVECTOR` implementation in the CUDA language. The module allows for SUNDIALS vector kernels to run on 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 class `Vector` in namespace `suncudavec` manages vector data layout:

```
template <class T, class I>
class Vector {
    I size_;
    I mem_size_;
    T* h_vec_;
    T* d_vec_;
    ThreadPartitioning<T, I>* partStream_;
    ThreadPartitioning<T, I>* partReduce_;
    bool ownPartitioning_;

    ...
};
```

The class members are vector size (length), size of the vector data memory block, pointers to vector data on the host and the device, pointers to `ThreadPartitioning` implementations that handle thread partitioning for streaming and reduction vector kernels, and a boolean flag that signals if the vector owns the thread partitioning. The class `Vector` inherits from the empty structure

```
struct _N_VectorContent_Cuda {
};
```

to interface the C++ class with the NVECTOR C code. When instantiated, the class `Vector` will allocate memory on both the host and the device. Due to the rapid progress of CUDA development, we expect that the `suncudavec::Vector` class will change frequently in future SUNDIALS releases. The code is structured so that it can tolerate significant changes in the `suncudavec::Vector` class without requiring changes to the user API.

The NVECTOR_CUDA module can be utilized for single-node parallelism or in a distributed context with MPI. The header file to include when using this module for single-node parallelism is `nvector_cuda.h`. The header file to include when using this module in the distributed case is `nvector_mpicuda.h`. Note that only the NVECTOR_CUDA constructor signature differs between the two header files. The installed module libraries to link to are `libsundials_nveccuda.lib` in the single-node case, or `libsundials_nvecmpicuda.lib` in the distributed case. Only one of these libraries may be linked to when creating an executable or library. SUNDIALS must be built with MPI support if the distributed library is desired. The extension, `.lib`, is typically `.so` for shared libraries and `.a` for static libraries.

Unlike other native SUNDIALS vector types, NVECTOR_CUDA does not provide macros to access its member variables. Instead, user should use the accessor functions in the namespace `suncudavec`.

- `getDevData(N_Vector v)`

This function takes an `N_Vector` as an argument and returns a raw pointer to the vector data on the device (GPU). It is the user's responsibility to ensure that the vector argument is of the correct `N_Vector` type.

- `getHostData(N_Vector v)`

This function takes a `N_Vector` as an argument and returns a raw pointer to the vector data on the host (CPU memory). It is the user's responsibility to ensure that the vector argument is of the correct `N_Vector` type.

- `getSize(N_Vector v)`

Returns the vector's local length.

- `getGlobalSize(N_Vector v)`

Returns the vector's global length.

- `getMPIComm(N_Vector v)`

Takes a `N_Vector` as an argument and returns a sundials communicator of type

The NVECTOR_CUDA module defines implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4, 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. Instead, the NVECTOR_CUDA module provides separate functions to access data on the host and on the device. It also provides methods for copying from the host to the device and vice versa. Usage examples of NVECTOR_CUDA are provided in some example programs for CVODE [27].

The names of vector operations are obtained from those in Tables 6.2, 6.3, and 6.4 by appending the suffix `_Cuda` (e.g. `N_VDestroy_Cuda`). The module NVECTOR_CUDA provides the following additional user-callable routines:

- **N_VNew_Cuda**

Note: this function signature is defined in the header `nvector_mpicuda.h` and should be used when using this module in a distributed context. This function creates and allocates memory for a CUDA `N_Vector`. The memory is allocated on both host and device. Its arguments are local and global vector lengths, as well as the MPI communicator. Use this constructor with the `libsundials_nvecmpicuda.lib` library.

```
N_Vector N_VNew_Cuda(MPI_Comm comm,
                     sunindextype local_length,
                     sunindextype global_length);
```

- **N_VNew_Cuda**

Note: this function signature is defined in the header `nvector_cuda.h` and should be used when using this module for single-node parallelism. This function creates and allocates memory for a CUDA `N_Vector` on a single node. The memory is allocated on both host and device. Its only argument is vector length. Use this constructor with the `libsundials_nveccuda.lib` library.

```
N_Vector N_VNew_Cuda(sunindextype length);
```

- **N_VNewEmpty_Cuda**

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_Vector N_VNewEmpty_Cuda(sunindextype vec_length);
```

- **N_VMake_Cuda**

This function creates and allocates memory for an NVECTOR_CUDA wrapper around a user-provided `suncudavec::Vector` class. Its only argument is of type `N_VectorContent_Cuda`, which is the pointer to the class.

```
N_Vector N_VMake_Cuda(N_VectorContent_Cuda c);
```

- **N_VGetLength_Cuda**

This function returns the length of the vector.

```
sunindextype N_VGetLength_Cuda(N_Vector v);
```

- **N_VGetHostArrayPointer_Cuda**

This function returns a pointer to the vector data on the host.

```
realtype *N_VGetHostArrayPointer_Cuda(N_Vector v);
```

- **N_VGetDeviceArrayPointer_Cuda**

This function returns a pointer to the vector data on the device.

```
realtype *N_VGetDeviceArrayPointer_Cuda(N_Vector v);
```

- **N_VCopyToDevice_Cuda**

This function copies host vector data to the device.

```
realtype *N_VCopyToDevice_Cuda(N_Vector v);
```

- **N_VCopyFromDevice_Cuda**

This function copies vector data from the device to the host.

```
realtype *N_VCopyFromDevice_Cuda(N_Vector v);
```

- `N_VPrint_Cuda`

This function prints the content of a CUDA vector to `stdout`.

```
void N_VPrint_Cuda(N_Vector v);
```

- `N_VPrintFile_Cuda`

This function prints the content of a CUDA vector to `outfile`.

```
void N_VPrintFile_Cuda(N_Vector v, FILE *outfile);
```

Notes

- When there is a need to access components of an `N_Vector_Cuda`, `v`, it is recommended to use functions `N_VGetDeviceArrayPointer_Cuda` or `N_VGetHostArrayPointer_Cuda`.



- To maximize efficiency, vector operations in the `NVECTOR_CUDA` implementation that have more than one `N_Vector` argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

6.8 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 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 a CUDA software development toolkit. Besides the CUDA backend, `RAJA` has other backends such as serial, OpenMP, and OpenAC. These backends are not used in this `SUNDIALS` release. Class `Vector` in namespace `sunrajavec` manages the vector data layout:

```
template <class T, class I>
class Vector {
    I size_;
    I mem_size_;
    T* h_vec_;
    T* d_vec_;
    ...
};
```

The class members are: vector size (length), size of the vector data memory block, and pointers to vector data on the host and on the device. The class `Vector` inherits from an empty structure

```
struct _N_VectorContent_Raja {
};
```

to interface the C++ class with the `NVECTOR` C code. When instantiated, the class `Vector` will allocate memory on both the host and the device. Due to the rapid progress of `RAJA` development, we expect that the `sunrajavec::Vector` class will change frequently in future `SUNDIALS` releases. The code is structured so that it can tolerate significant changes in the `sunrajavec::Vector` class without requiring changes to the user API.

The `NVECTOR_RAJA` module can be utilized for single-node parallelism or in a distributed context with MPI. The header file to include when using this module for single-node parallelism is `nvector_raja.h`. The header file to include when using this module in the distributed case is `nvector_mpiraja.h`. Note that only the `NVECTOR_RAJA` constructor signature differs between the two header files. The installed module libraries to link to are `libsundials_nvecraja.lib` in the single-node case, or `libsundials_nvecmpicudaraja.lib` in the distributed case. Only one one of

these libraries may be linked to when creating an executable or library. SUNDIALS must be built with MPI support if the distributed library is desired. The extension, *.lib*, is typically *.so* for shared libraries and *.a* for static libraries.

Unlike other native SUNDIALS vector types, NVECTOR_RAJA does not provide macros to access its member variables. Instead, user should use the accessor functions in the namespace `sunraja_vec`.

- `getDevData(N_Vector v)`

This function takes a `N_Vector` as an argument and returns a raw pointer to the vector data on the device (GPU). It is the user's responsibility to ensure that the vector argument is of the correct `N_Vector` type.

- `getHostData(N_Vector v)`

This function takes a `N_Vector` as an argument and returns a raw pointer to the vector data on the host (CPU memory). It is the user's responsibility to ensure that the vector argument is of the correct `N_Vector` type.

- `getSize(N_Vector v)`

Returns the vector's local length.

- `getGlobalSize(N_Vector v)`

Returns the vector's global length.

- `getMPIComm(N_Vector v)`

Takes a `N_Vector` as an argument and returns a sundials communicator of type `SUNDIALS_Comm`.

The NVECTOR_RAJA module defines the implementations of all vector operations listed in Tables 6.2, 6.3, and 6.4, 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 implemented by the RAJA vector. 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 [27].

The names of vector operations are obtained from those in Tables 6.2, 6.3, and 6.4, by appending the suffix `_Raja` (e.g. `N_VDestroy_Raja`). The module NVECTOR_RAJA provides the following additional user-callable routines:

- `N_VNew_Raja`

Note: this function signature is defined in the header `nvector_mpiraja.h` and should be used when using this module in a distributed context. This function creates and allocates memory for a RAJA `N_Vector`. The memory is allocated on both host and device. Its arguments are local and global vector lengths, as well as the MPI communicator. Use this constructor with the `libsundials_nvecmpicudaraja.lib` library.

```
N_Vector N_VNew_Raja(MPI_Comm comm,
                     sunindextype local_length,
                     sunindextype global_length);
```

- `N_VNew_Raja`

Note: this function signature is defined in the header `nvector_raja.h` and should be used when using this module for single-node parallelism. This function creates and allocates memory for a RAJA `N_Vector` on a single node. The memory is allocated on both host and device. Its only argument is vector length. Use this constructor with the `libsundials_nveccudaraja.lib` library.

```
N_Vector N_VNew_Raja(sunindextype length);
```

- **N_VNewEmpty_Raja**

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_Vector N_VNewEmpty_Raja(sunindextype vec_length);
```

- **N_VMake_Raja**

This function creates and allocates memory for an NVECTOR_RAJA wrapper around a user-provided `sunrajavec::Vector` class. Its only argument is of type `N_VectorContent_Raja`, which is the pointer to the class.

```
N_Vector N_VMake_Raja(N_VectorContent_Raja c);
```

- **N_VGetLength_Raja**

This function returns the length of the vector.

```
sunindextype N_VGetLength_Raja(N_Vector v);
```

- **N_VGetHostArrayPointer_Raja**

This function returns a pointer to the vector data on the host.

```
realtype *N_VGetHostArrayPointer_Raja(N_Vector v);
```

- **N_VGetDeviceArrayPointer_Raja**

This function returns a pointer to the vector data on the device.

```
realtype *N_VGetDeviceArrayPointer_Raja(N_Vector v);
```

- **N_VCopyToDevice_Raja**

This function copies host vector data to the device.

```
realtype *N_VCopyToDevice_Raja(N_Vector v);
```

- **N_VCopyFromDevice_Raja**

This function copies vector data from the device to the host.

```
realtype *N_VCopyFromDevice_Raja(N_Vector v);
```

- **N_VPrint_Raja**

This function prints the content of a RAJA vector to `stdout`.

```
void N_VPrint_Raja(N_Vector v);
```

- **N_VPrintFile_Raja**

This function prints the content of a RAJA vector to `outfile`.

```
void N_VPrintFile_Raja(N_Vector v, FILE *outfile);
```

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`.



- To maximize efficiency, vector operations in the NVECTOR_RAJA implementation that have more than one `N_Vector` argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

6.9 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_VLinearSum` Case 1a: Test $y = x + y$
- `Test_N_VLinearSum` Case 1b: Test $y = -x + y$
- `Test_N_VLinearSum` Case 1c: Test $y = ax + y$
- `Test_N_VLinearSum` Case 2a: Test $x = x + y$
- `Test_N_VLinearSum` Case 2b: Test $x = x - y$
- `Test_N_VLinearSum` Case 2c: Test $x = x + by$
- `Test_N_VLinearSum` Case 3: Test $z = x + y$
- `Test_N_VLinearSum` Case 4a: Test $z = x - y$
- `Test_N_VLinearSum` Case 4b: Test $z = -x + y$
- `Test_N_VLinearSum` Case 5a: Test $z = x + by$
- `Test_N_VLinearSum` Case 5b: Test $z = ax + y$
- `Test_N_VLinearSum` Case 6a: Test $z = -x + by$
- `Test_N_VLinearSum` Case 6b: Test $z = ax - y$
- `Test_N_VLinearSum` Case 7: Test $z = a(x + y)$
- `Test_N_VLinearSum` Case 8: Test $z = a(x - y)$
- `Test_N_VLinearSum` Case 9: Test $z = ax + by$
- `Test_N_VConst`: Fill vector with constant and check result.
- `Test_N_VProd`: Test vector multiply: $z = x * y$
- `Test_N_VDiv`: Test vector division: $z = x / y$
- `Test_N_VScale`: Case 1: scale: $x = cx$
- `Test_N_VScale`: Case 2: copy: $z = x$

- `Test_N_VScale`: Case 3: negate: $z = -x$
- `Test_N_VScale`: Case 4: combination: $z = cx$
- `Test_N_VAbs`: Create absolute value of vector.
- `Test_N_VAddConst`: add constant vector: $z = c + x$
- `Test_N_VDotProd`: Calculate dot product of two vectors.
- `Test_N_VMaxNorm`: Create vector with known values, find and validate the max norm.
- `Test_N_VWrmsNorm`: Create vector of known values, find and validate the weighted root mean square.
- `Test_N_VWrmsNormMask`: Create vector of known values, find and validate the weighted root mean square using all elements except one.
- `Test_N_VMin`: Create vector, find and validate the min.
- `Test_N_VWL2Norm`: Create vector, find and validate the weighted Euclidean L2 norm.
- `Test_N_VL1Norm`: Create vector, find and validate the L1 norm.
- `Test_N_VCompare`: Compare vector with constant returning and validating comparison vector.
- `Test_N_VInvTest`: Test $z[i] = 1 / x[i]$
- `Test_N_VConstrMask`: Test mask of vector x with vector c .
- `Test_N_VMinQuotient`: Fill two vectors with known values. Calculate and validate minimum quotient.
- `Test_N_VLinearCombination` Case 1a: Test $x = a x$
- `Test_N_VLinearCombination` Case 1b: Test $z = a x$
- `Test_N_VLinearCombination` Case 2a: Test $x = a x + b y$
- `Test_N_VLinearCombination` Case 2b: Test $z = a x + b y$
- `Test_N_VLinearCombination` Case 3a: Test $x = x + a y + b z$
- `Test_N_VLinearCombination` Case 3b: Test $x = a x + b y + c z$
- `Test_N_VLinearCombination` Case 3c: Test $w = a x + b y + c z$
- `Test_N_VScaleAddMulti` Case 1a: $y = a x + y$
- `Test_N_VScaleAddMulti` Case 1b: $z = a x + y$
- `Test_N_VScaleAddMulti` Case 2a: $Y[i] = c[i] x + Y[i]$, $i = 1,2,3$
- `Test_N_VScaleAddMulti` Case 2b: $Z[i] = c[i] x + Y[i]$, $i = 1,2,3$
- `Test_N_VDotProdMulti` Case 1: Calculate the dot product of two vectors
- `Test_N_VDotProdMulti` Case 2: Calculate the dot product of one vector with three other vectors in a vector array.
- `Test_N_VLinearSumVectorArray` Case 1: $z = a x + b y$
- `Test_N_VLinearSumVectorArray` Case 2a: $Z[i] = a X[i] + b Y[i]$
- `Test_N_VLinearSumVectorArray` Case 2b: $X[i] = a X[i] + b Y[i]$

- Test_N_VLinearSumVectorArray Case 2c: $Y[i] = a X[i] + b Y[i]$
- Test_N_VScaleVectorArray Case 1a: $y = c y$
- Test_N_VScaleVectorArray Case 1b: $z = c y$
- Test_N_VScaleVectorArray Case 2a: $Y[i] = c[i] Y[i]$
- Test_N_VScaleVectorArray Case 2b: $Z[i] = c[i] Y[i]$
- Test_N_VScaleVectorArray Case 1a: $z = c$
- Test_N_VScaleVectorArray Case 1b: $Z[i] = c$
- Test_N_VWrmsNormVectorArray Case 1a: Create a vector of know values, find and validate the weighted root mean square norm.
- Test_N_VWrmsNormVectorArray Case 1b: Create a vector array of three vectors of know values, find and validate the weighted root mean square norm of each.
- Test_N_VWrmsNormMaskVectorArray Case 1a: Create a vector of know values, find and validate the weighted root mean square norm using all elements except one.
- Test_N_VWrmsNormMaskVectorArray Case 1b: Create a vector array of three vectors of know values, find and validate the weighted root mean square norm of each using all elements except one.
- Test_N_VScaleAddMultiVectorArray Case 1a: $y = a x + y$
- Test_N_VScaleAddMultiVectorArray Case 1b: $z = a x + y$
- Test_N_VScaleAddMultiVectorArray Case 2a: $Y[j][0] = a[j] X[0] + Y[j][0]$
- Test_N_VScaleAddMultiVectorArray Case 2b: $Z[j][0] = a[j] X[0] + Y[j][0]$
- Test_N_VScaleAddMultiVectorArray Case 3a: $Y[0][i] = a[0] X[i] + Y[0][i]$
- Test_N_VScaleAddMultiVectorArray Case 3b: $Z[0][i] = a[0] X[i] + Y[0][i]$
- Test_N_VScaleAddMultiVectorArray Case 4a: $Y[j][i] = a[j] X[i] + Y[j][i]$
- Test_N_VScaleAddMultiVectorArray Case 4b: $Z[j][i] = a[j] X[i] + Y[j][i]$
- Test_N_VLinearCombinationVectorArray Case 1a: $x = a x$
- Test_N_VLinearCombinationVectorArray Case 1b: $z = a x$
- Test_N_VLinearCombinationVectorArray Case 2a: $x = a x + b y$
- Test_N_VLinearCombinationVectorArray Case 2b: $z = a x + b y$
- Test_N_VLinearCombinationVectorArray Case 3a: $x = a x + b y + c z$
- Test_N_VLinearCombinationVectorArray Case 3b: $w = a x + b y + c z$
- Test_N_VLinearCombinationVectorArray Case 4a: $X[0][i] = c[0] X[0][i]$
- Test_N_VLinearCombinationVectorArray Case 4b: $Z[i] = c[0] X[0][i]$
- Test_N_VLinearCombinationVectorArray Case 5a: $X[0][i] = c[0] X[0][i] + c[1] X[1][i]$
- Test_N_VLinearCombinationVectorArray Case 5b: $Z[i] = c[0] X[0][i] + c[1] X[1][i]$
- Test_N_VLinearCombinationVectorArray Case 6a: $X[0][i] = X[0][i] + c[1] X[1][i] + c[2] X[2][i]$
- Test_N_VLinearCombinationVectorArray Case 6b: $X[0][i] = c[0] X[0][i] + c[1] X[1][i] + c[2] X[2][i]$
- Test_N_VLinearCombinationVectorArray Case 6c: $Z[i] = c[0] X[0][i] + c[1] X[1][i] + c[2] X[2][i]$

6.10 NVECTOR functions used by CVODE

In Table 6.5 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.

Table 6.5: List of vector functions usage by CVODE code modules

	CVODE	CVLS	CVDIAG	CVBANDPRE	CVBBDPRE	FCVODE
N_VGetVectorID						
N_VClone	✓	✓	✓			
N_VCloneEmpty		1				✓
N_VDestroy	✓	✓	✓			
N_VSpace	✓	2				
N_VGetArrayPointer		1		✓	✓	✓
N_VSetArrayPointer		1				✓
N_VLinearSum	✓	✓	✓			
N_VConst	✓	✓				
N_VProd	✓		✓			
N_VDiv	✓		✓			
N_VScale	✓	✓	✓	✓	✓	
N_VAbs	✓					
N_VInv	✓		✓			
N_VAddConst	✓		✓			
N_VDotProd		✓				
N_VMaxNorm	✓					
N_VWrmsNorm	✓	✓		✓	✓	
N_VMin	✓					
N_VMinQuotient	✓					
N_VConstrMask	✓					
N_VCompare	✓		✓			
N_VInvTest			✓			
N_VLinearCombination	✓					
N_VScaleAddMulti	✓					
N_VDotProdMulti	3	3				
N_VScaleVectorArray	✓					

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 Tables 6.3 and 6.4 not listed above are unused and a user-supplied NVECTOR module for CVODE could omit these operations.

Each `SUNLINSOL` object may require additional NVECTOR routines not listed in the table above. Please see the relevant descriptions of these modules in Sections 8.3-8.13 for additional detail on their NVECTOR requirements.

The vector functions listed in Table 6.2 that are *not* used by CVODE are: `N_VWL2Norm`, `N_VL1Norm`, and `N_VWrmsNormMask`. Therefore, a user-supplied NVECTOR module for CVODE could omit these functions. 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.

Chapter 7

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.

The generic `SUNMatrix` type has been modeled after the object-oriented style of the generic `N_Vector` type. Specifically, a generic `SUNMatrix` is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the matrix, and an *ops* field pointing to a structure with generic matrix operations. The type `SUNMatrix` is defined as

```
typedef struct _generic_SUNMatrix *SUNMatrix;
```

```
struct _generic_SUNMatrix {  
    void *content;  
    struct _generic_SUNMatrix_Ops *ops;  
};
```

The `_generic_SUNMatrix_Ops` structure is essentially a list of pointers to the various actual matrix operations, and is defined as

```
struct _generic_SUNMatrix_Ops {  
    SUNMatrix_ID (*getid)(SUNMatrix);  
    SUNMatrix (*clone)(SUNMatrix);  
    void (*destroy)(SUNMatrix);  
    int (*zero)(SUNMatrix);  
    int (*copy)(SUNMatrix, SUNMatrix);  
    int (*scaleadd)(realtype, SUNMatrix, SUNMatrix);  
    int (*scaleaddi)(realtype, SUNMatrix);  
    int (*matvec)(SUNMatrix, N_Vector, N_Vector);  
    int (*space)(SUNMatrix, long int*, long int*);  
};
```

The generic SUNMATRIX module defines and implements the matrix operations acting on `SUNMatrix` objects. These routines are nothing but wrappers for the matrix operations defined by a particular SUNMATRIX implementation, which are accessed through the *ops* field of the `SUNMatrix` structure. To

Table 7.1: Identifiers associated with matrix kernels supplied with SUNDIALS.

Matrix ID	Matrix type	ID Value
SUNMATRIX_DENSE	Dense $M \times N$ matrix	0
SUNMATRIX_BAND	Band $M \times M$ matrix	1
SUNMATRIX_SPARSE	Sparse (CSR or CSC) $M \times N$ matrix	2
SUNMATRIX_CUSTOM	User-provided custom matrix	3

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));
}
```

Table 7.2 contains a complete list of all matrix operations defined by the generic SUNMATRIX module. A particular implementation of the SUNMATRIX module must:

- Specify the *content* field of the `SUNMatrix` object.
- Define and implement a minimal subset of the matrix operations. See the documentation for each SUNDIALS 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`.

Each SUNMATRIX implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in Table 7.1. It is recommended that a user-supplied SUNMATRIX implementation use the `SUNMATRIX_CUSTOM` identifier.

Table 7.2: Description of the `SUNMatrix` operations

Name	Usage and Description
SUNMatGetID	<code>id = SUNMatGetID(A);</code> Returns the type identifier for the matrix <code>A</code> . It is used to determine the matrix implementation type (e.g. dense, banded, sparse, . . .) from the abstract <code>SUNMatrix</code> interface. This is used to assess compatibility with SUNDIALS-provided linear solver implementations. Returned values are given in the Table 7.1.
<i>continued on next page</i>	

Name	Usage and Description
SUNMatClone	<pre>B = SUNMatClone(A);</pre> <p>Creates a new SUNMatrix of the same type as an existing matrix A and sets the <i>ops</i> field. It does not copy the matrix, but rather allocates storage for the new matrix.</p>
SUNMatDestroy	<pre>SUNMatDestroy(A);</pre> <p>Destroys the SUNMatrix A and frees memory allocated for its internal data.</p>
SUNMatSpace	<pre>ier = SUNMatSpace(A, &lrw, &liw);</pre> <p>Returns the storage requirements for the matrix A. lrw is a long int containing the number of realtype words and liw is a long int containing the number of integer words. The return value is an integer flag denoting success/failure of the operation.</p> <p>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.</p>
SUNMatZero	<pre>ier = SUNMatZero(A);</pre> <p>Performs the operation $A_{ij} = 0$ for all entries of the matrix <i>A</i>. The return value is an integer flag denoting success/failure of the operation.</p>
SUNMatCopy	<pre>ier = SUNMatCopy(A,B);</pre> <p>Performs the operation $B_{ij} = A_{i,j}$ for all entries of the matrices <i>A</i> and <i>B</i>. The return value is an integer flag denoting success/failure of the operation.</p>
SUNMatScaleAdd	<pre>ier = SUNMatScaleAdd(c, A, B);</pre> <p>Performs the operation $A = cA + B$. The return value is an integer flag denoting success/failure of the operation.</p>
SUNMatScaleAddI	<pre>ier = SUNMatScaleAddI(c, A);</pre> <p>Performs the operation $A = cA + I$. The return value is an integer flag denoting success/failure of the operation.</p>
SUNMatMatvec	<pre>ier = SUNMatMatvec(A, x, y);</pre> <p>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. The return value is an integer flag denoting success/failure of the operation.</p>

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 8. More specifically, in Table 7.3 we show the matrix interfaces available as `SUNMATRIX` modules, and the compatible vector implementations.

Table 7.3: SUNDIALS matrix interfaces and vector implementations that can be used for each.

Matrix Interface	Serial	Parallel (MPI)	OpenMP	pThreads	<i>hypr</i> Vec.	PETSc Vec.	CUDA	RAJA	User Suppl.
Dense	✓		✓	✓					✓

continued on next page

continued on next page

Matrix Interface	Serial	Parallel (MPI)	OpenMP	pThreads	hybre Vec.	PETSc Vec.	CUDA	RAJA	User Suppl.
Band	✓		✓	✓					✓
Sparse	✓		✓	✓					✓
User supplied	✓	✓	✓	✓	✓	✓	✓	✓	✓

7.1 The SUNMatrix_Dense implementation

The dense implementation of the SUNMATRIX module provided with SUNDIALS, SUNMATRIX_DENSE, defines the *content* field of **SUNMatrix** to be the following structure:

```
struct _SUNMatrixContent_Dense {
    sunindextype M;
    sunindextype N;
    realtype *data;
    sunindextype ldata;
    realtype **cols;
};
```

These entries of the *content* field contain the following information:

M - number of rows

N - number of columns

data - pointer to a contiguous block of **realtype** variables. The elements of the dense matrix are stored columnwise, i.e. the (i,j)-th element of a dense SUNMATRIX **A** (with $0 \leq i < M$ and $0 \leq j < N$) may be accessed via **data[j*M+i]**.

ldata - length of the data array (= M·N).

cols - array of pointers. **cols[j]** points to the first element of the j-th column of the matrix in the array **data**. The (i,j)-th element of a dense SUNMATRIX **A** (with $0 \leq i < M$ and $0 \leq j < N$) may be accessed via **cols[j][i]**.

The header file to include when using this module is **sunmatrix/sunmatrix_dense.h**. The SUNMATRIX_DENSE module is accessible from all SUNDIALS solvers *without* linking to the **libsundials_sunmatrixdense** module library.

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_LDATAL_D**

These macros give individual access to various lengths relevant to the content of a dense **SUNMatrix**.

These may be used either to retrieve or to set these values. For example, the assignment `A_rows = SM_ROWS_D(A)` sets `A_rows` to be the number of rows in the matrix `A`. Similarly, the assignment `SM_COLUMNS_D(A) = A_cols` sets the number of columns in `A` to equal `A_cols`.

Implementation:

```
#define SM_ROWS_D(A)      ( SM_CONTENT_D(A)->M )
#define SM_COLUMNS_D(A)   ( SM_CONTENT_D(A)->N )
#define SM_LDATA_D(A)     ( SM_CONTENT_D(A)->ldata )
```

- `SM_DATA_D` and `SM_COLS_D`

These macros give access to the `data` and `cols` pointers for the matrix entries.

The assignment `A_data = SM_DATA_D(A)` sets `A_data` to be a pointer to the first component of the data array for the dense `SUNMatrix` `A`. The assignment `SM_DATA_D(A) = A_data` sets the data array of `A` to be `A_data` by storing the pointer `A_data`.

Similarly, the assignment `A_cols = SM_COLS_D(A)` sets `A_cols` to be a pointer to the array of column pointers for the dense `SUNMatrix` `A`. The assignment `SM_COLS_D(A) = A_cols` sets the column pointer array of `A` to be `A_cols` by storing the pointer `A_cols`.

Implementation:

```
#define SM_DATA_D(A)      ( SM_CONTENT_D(A)->data )
#define SM_COLS_D(A)      ( SM_CONTENT_D(A)->cols )
```

- `SM_COLUMN_D` and `SM_ELEMENT_D`

These macros give access to the individual columns and entries of the data array of a dense `SUNMatrix`.

The assignment `col_j = SM_COLUMN_D(A,j)` sets `col_j` to be a pointer to the first entry of the j -th column of the $M \times N$ dense matrix `A` (with $0 \leq j < N$). The type of the expression `SM_COLUMN_D(A,j)` is `realtype *`. The pointer returned by the call `SM_COLUMN_D(A,j)` can be treated as an array which is indexed from 0 to $M - 1$.

The assignments `SM_ELEMENT_D(A,i,j) = a_ij` and `a_ij = SM_ELEMENT_D(A,i,j)` reference the (i,j) -th element of the $M \times N$ dense matrix `A` (with $0 \leq i < M$ and $0 \leq j < N$).

Implementation:

```
#define SM_COLUMN_D(A,j)   ( (SM_CONTENT_D(A)->cols)[j] )
#define SM_ELEMENT_D(A,i,j) ( (SM_CONTENT_D(A)->cols)[j][i] )
```

The `SUNMATRIX_DENSE` module defines dense implementations of all matrix operations listed in Table 7.2. Their names are obtained from those in Table 7.2 by appending the suffix `_Dense` (e.g. `SUNMatCopy_Dense`). The module `SUNMATRIX_DENSE` provides the following additional user-callable routines:

- `SUNDenseMatrix`

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.

```
SUNMatrix SUNDenseMatrix(sunindextype M, sunindextype N);
```

- `SUNDenseMatrix.Print`

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.

```
void SUNDenseMatrix_Print(SUNMatrix A, FILE* outfile);
```

- **SUNDenseMatrix.Rows**
This function returns the number of rows in the dense **SUNMatrix**.
`sunindextype SUNDenseMatrix_Rows(SUNMatrix A);`
- **SUNDenseMatrix.Columns**
This function returns the number of columns in the dense **SUNMatrix**.
`sunindextype SUNDenseMatrix_Columns(SUNMatrix A);`
- **SUNDenseMatrix.LData**
This function returns the length of the data array for the dense **SUNMatrix**.
`sunindextype SUNDenseMatrix_LData(SUNMatrix A);`
- **SUNDenseMatrix.Data**
This function returns a pointer to the data array for the dense **SUNMatrix**.
`realtype* SUNDenseMatrix_Data(SUNMatrix A);`
- **SUNDenseMatrix.Cols**
This function returns a pointer to the cols array for the dense **SUNMatrix**.
`realtype** SUNDenseMatrix_Cols(SUNMatrix A);`
- **SUNDenseMatrix.Column**
This function returns a pointer to the first entry of the *j*th column of the dense **SUNMatrix**. The resulting pointer should be indexed over the range 0 to *M* − 1.
`realtype* SUNDenseMatrix_Column(SUNMatrix A, sunindextype j);`

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.

For solvers that include a Fortran interface module, the **SUNMATRIX_DENSE** module also includes the Fortran-callable function **FSUNDenseMatInit**(*code*, *M*, *N*, *ier*) to initialize this **SUNMATRIX_DENSE** module for a given **SUNDIALS** solver. Here *code* is an integer input solver id (1 for **CVODE**, 2 for **IDA**, 3 for **KINSOL**, 4 for **ARKODE**); *M* and *N* are the corresponding dense matrix construction arguments (declared to match C type **long int**); and *ier* is an error return flag equal to 0 for success and -1 for failure. Both *code* and *ier* are declared to match C type **int**. Additionally, when using **ARKODE** with a non-identity mass matrix, the Fortran-callable function **FSUNDenseMassMatInit**(*M*, *N*, *ier*) initializes this **SUNMATRIX_DENSE** module for storing the mass matrix.

7.2 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 7.1. A more complete description of the parts of this *content* field is given below:

M - number of rows

N - number of columns ($N = M$)

mu - upper half-bandwidth, $0 \leq \mu < N$

ml - lower half-bandwidth, $0 \leq ml < N$

s_mu - storage upper bandwidth, $\mu \leq s_mu < N$. 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 ($ldim \geq s_mu+ml+1$)

data - pointer to a contiguous block of **realtype** variables. The elements of the banded matrix are stored columnwise (i.e. columns are stored one on top of the other in memory). Only elements within the specified half-bandwidths are stored. **data** is a pointer to **ldata** contiguous locations which hold the elements within the band of A.

ldata - length of the data array ($= ldim \cdot N$)

cols - array of pointers. **cols**[j] is a pointer to the uppermost element within the band in the j-th column. This pointer may be treated as an array indexed from **s_mu**-**mu** (to access the uppermost element within the band in the j-th column) to **s_mu**+**ml** (to access the lowest element within the band in the j-th column). Indices from 0 to **s_mu**-**mu**-1 give access to extra storage elements required by the LU decomposition function. Finally, **cols**[j][i-j+s_mu] is the (i,j)-th element with $j-\mu \leq i \leq j+ml$.

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.

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.

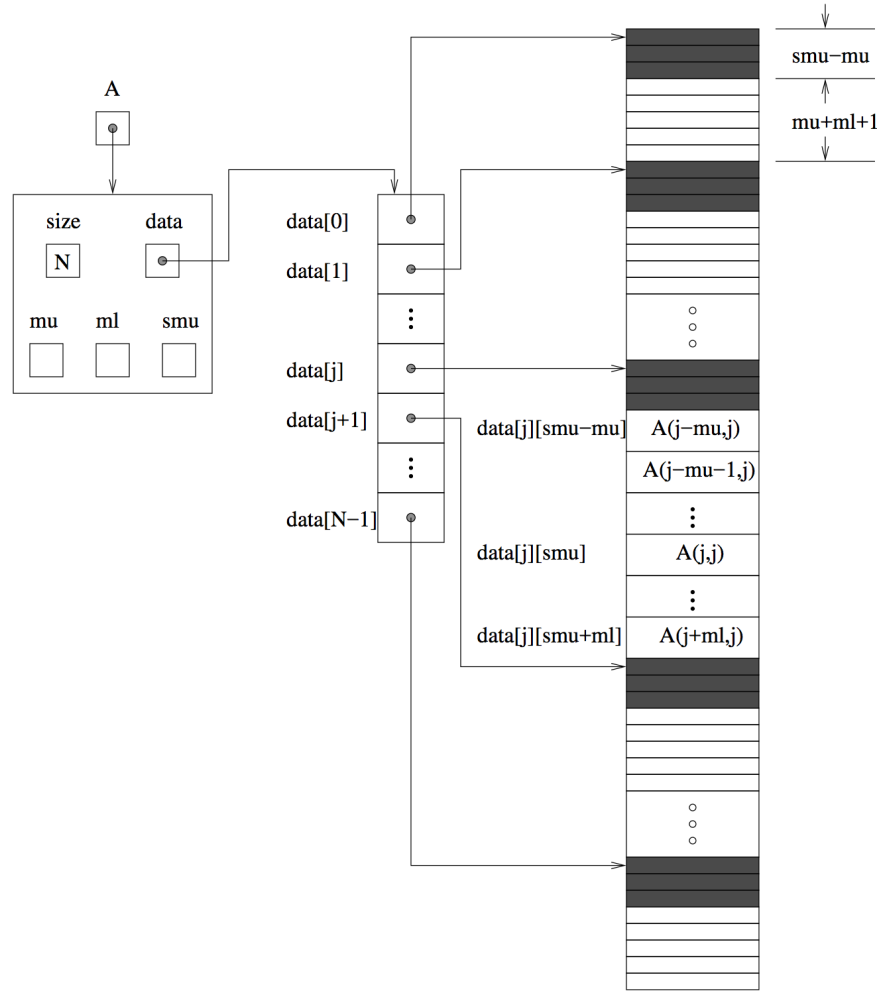


Figure 7.1: Diagram of the storage for the `SUNMATRIX_BAND` module. Here A is an $N \times N$ band matrix with upper and lower half-bandwidths μ and ml , respectively. The rows and columns of A are numbered from 0 to $N - 1$ and the (i, j) -th element of A is denoted $A(i, j)$. The greyed out areas of the underlying component storage are used by the associated `SUNLINSOL_BAND` linear solver.

- **SM_CONTENT_B**

This routine gives access to the contents of the banded **SUNMatrix**.

The assignment **A_cont = SM_CONTENT_B(A)** sets **A_cont** to be a pointer to the banded **SUNMatrix** content structure.

Implementation:

```
#define SM_CONTENT_B(A)      ( (SUNMatrixContent_Band)(A->content) )
```

- **SM_ROWS_B**, **SM_COLUMNS_B**, **SM_UBAND_B**, **SM_LBAND_B**, **SM_SUBAND_B**, **SM_LDIM_B**, and **SM_LDATA_B**

These macros give individual access to various lengths relevant to the content of a banded **SUNMatrix**.

These may be used either to retrieve or to set these values. For example, the assignment **A_rows = SM_ROWS_B(A)** sets **A_rows** to be the number of rows in the matrix **A**. Similarly, the assignment **SM_COLUMNS_B(A) = A_cols** sets the number of columns in **A** to equal **A_cols**.

Implementation:

```
#define SM_ROWS_B(A)         ( SM_CONTENT_B(A)->M )
#define SM_COLUMNS_B(A)      ( SM_CONTENT_B(A)->N )
#define SM_UBAND_B(A)        ( SM_CONTENT_B(A)->mu )
#define SM_LBAND_B(A)        ( SM_CONTENT_B(A)->m1 )
#define SM_SUBAND_B(A)        ( SM_CONTENT_B(A)->s_mu )
#define SM_LDIM_B(A)         ( SM_CONTENT_B(A)->ldim )
#define SM_LDATA_B(A)        ( SM_CONTENT_B(A)->ldata )
```

- **SM_DATA_B** and **SM_COLS_B**

These macros give access to the **data** and **cols** pointers for the matrix entries.

The assignment **A_data = SM_DATA_B(A)** sets **A_data** to be a pointer to the first component of the data array for the banded **SUNMatrix** **A**. The assignment **SM_DATA_B(A) = A_data** sets the data array of **A** to be **A_data** by storing the pointer **A_data**.

Similarly, the assignment **A_cols = SM_COLS_B(A)** sets **A_cols** to be a pointer to the array of column pointers for the banded **SUNMatrix** **A**. The assignment **SM_COLS_B(A) = A_cols** sets the column pointer array of **A** to be **A_cols** by storing the pointer **A_cols**.

Implementation:

```
#define SM_DATA_B(A)         ( SM_CONTENT_B(A)->data )
#define SM_COLS_B(A)         ( SM_CONTENT_B(A)->cols )
```

- **SM_COLUMN_B**, **SM_COLUMN_ELEMENT_B**, and **SM_ELEMENT_B**

These macros give access to the individual columns and entries of the data array of a banded **SUNMatrix**.

The assignments **SM_ELEMENT_B(A,i,j) = a_ij** and **a_ij = SM_ELEMENT_B(A,i,j)** reference the (i,j) -th element of the $N \times N$ band matrix **A**, where $0 \leq i, j \leq N - 1$. The location (i,j) should further satisfy $j - \mu \leq i \leq j + m1$.

The assignment **col_j = SM_COLUMN_B(A,j)** sets **col_j** to be a pointer to the diagonal element of the j -th column of the $N \times N$ band matrix **A**, $0 \leq j \leq N - 1$. The type of the expression **SM_COLUMN_B(A,j)** is **realtype ***. The pointer returned by the call **SM_COLUMN_B(A,j)** can be treated as an array which is indexed from $-\mu$ to $m1$.

The assignments **SM_COLUMN_ELEMENT_B(col_j,i,j) = a_ij** and **a_ij = SM_COLUMN_ELEMENT_B(col_j,i,j)** reference the (i,j) -th entry of the band matrix **A** when used in conjunction with **SM_COLUMN_B** to reference the j -th column through **col_j**. The index (i,j) should satisfy $j - \mu \leq i \leq j + m1$.

Implementation:

```
#define SM_COLUMN_B(A,j)      ( ((SM_CONTENT_B(A)->cols)[j])+SM_SUBBAND_B(A) )
#define SM_COLUMN_ELEMENT_B(col_j,i,j) (col_j[(i)-(j)])
#define SM_ELEMENT_B(A,i,j)
    ( (SM_CONTENT_B(A)->cols)[j][(i)-(j)+SM_SUBBAND_B(A)] )
```

The `SUNMATRIX_BAND` module defines banded implementations of all matrix operations listed in Table 7.2. Their names are obtained from those in Table 7.2 by appending the suffix `_Band` (e.g. `SUNMatCopy_Band`). The module `SUNMATRIX_BAND` provides the following additional user-callable routines:

- **SUNBandMatrix**

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 μ if used in some other manner.

```
SUNMatrix SUNBandMatrix(sunindextype N, sunindextype mu,
                        sunindextype ml, sunindextype smu);
```

- **SUNBandMatrix.Print**

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.

```
void SUNBandMatrix_Print(SUNMatrix A, FILE* outfile);
```

- **SUNBandMatrix.Rows**

This function returns the number of rows in the banded **SUNMatrix**.

```
sunindextype SUNBandMatrix_Rows(SUNMatrix A);
```

- **SUNBandMatrix.Columns**

This function returns the number of columns in the banded **SUNMatrix**.

```
sunindextype SUNBandMatrix_Columns(SUNMatrix A);
```

- **SUNBandMatrix.LowerBandwidth**

This function returns the lower half-bandwidth of the banded **SUNMatrix**.

```
sunindextype SUNBandMatrix_LowerBandwidth(SUNMatrix A);
```

- **SUNBandMatrix.UpperBandwidth**

This function returns the upper half-bandwidth of the banded **SUNMatrix**.

```
sunindextype SUNBandMatrix_UpperBandwidth(SUNMatrix A);
```

- **SUNBandMatrix.StoredUpperBandwidth**

This function returns the stored upper half-bandwidth of the banded **SUNMatrix**.

```
sunindextype SUNBandMatrix_StoredUpperBandwidth(SUNMatrix A);
```


- `SUNBandMatrix_LDim`

This function returns the length of the leading dimension of the banded `SUNMatrix`.

```
sunindextype SUNBandMatrix_LDim(SUNMatrix A);
```

- `SUNBandMatrix_Data`

This function returns a pointer to the data array for the banded `SUNMatrix`.

```
realtype* SUNBandMatrix_Data(SUNMatrix A);
```

- `SUNBandMatrix_Cols`

This function returns a pointer to the cols array for the banded `SUNMatrix`.

```
realtype** SUNBandMatrix_Cols(SUNMatrix A);
```

- `SUNBandMatrix_Column`

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 m .

```
realtype* SUNBandMatrix_Column(SUNMatrix A, sunindextype j);
```

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.



For solvers that include a Fortran interface module, the `SUNMATRIX_BAND` module also includes the Fortran-callable function `FSUNBandMatInit(code, N, mu, ml, smu, 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`, `ml` and `smu` 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, smu, ier)` initializes this `SUNMATRIX_BAND` module for storing the mass matrix.

7.3 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 7.2 (the CSR format is similar). A more complete description of the parts of this *content* field is given below:

M - number of rows

N - number of columns

NNZ - maximum number of nonzero entries in the matrix (allocated length of **data** and **indexvals** arrays)

NP - number of index pointers (e.g. number of column pointers for CSC matrix). For CSC matrices $NP = N$, and for CSR matrices $NP = M$. This value is set automatically based the input for **sparsetype**.

data - pointer to a contiguous block of **realtype** variables (of length **NNZ**), containing the values of the nonzero entries in the matrix

sparsetype - type of the sparse matrix (**CSC_MAT** or **CSR_MAT**)

indexvals - pointer to a contiguous block of **int** variables (of length **NNZ**), containing the row indices (if CSC) or column indices (if CSR) of each nonzero matrix entry held in **data**

indexptrs - pointer to a contiguous block of **int** variables (of length **NP+1**). For CSC matrices each entry provides the index of the first column entry into the **data** and **indexvals** arrays, e.g. if **indexptr[3]=7**, then the first nonzero entry in the fourth column of the matrix is located in **data[7]**, and is located in row **indexvals[7]** of the matrix. The last entry contains the total number of nonzero values in the matrix and hence points one past the end of the active data in the **data** and **indexvals** arrays. For CSR matrices, each entry provides the index of the first row entry into the **data** and **indexvals** arrays.

The following pointers are added to the **SlsMat** type for user convenience, to provide a more intuitive interface to the CSC and CSR sparse matrix data structures. They are set automatically when creating a sparse SUNMATRIX, based on the sparse matrix storage type.

rowvals - pointer to **indexvals** when **sparsetype** is **CSC_MAT**, otherwise set to **NULL**.

colptrs - pointer to **indexptrs** when **sparsetype** is **CSC_MAT**, otherwise set to **NULL**.

colvals - pointer to **indexvals** when **sparsetype** is **CSR_MAT**, otherwise set to **NULL**.

rowptrs - pointer to **indexptrs** when **sparsetype** is **CSR_MAT**, otherwise set to **NULL**.

For example, the 5×4 CSC matrix

$$\begin{bmatrix} 0 & 3 & 1 & 0 \\ 3 & 0 & 0 & 2 \\ 0 & 7 & 0 & 0 \\ 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 5 \end{bmatrix}$$

could be stored in this structure as either

```
M = 5;
N = 4;
NNZ = 8;
NP = N;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0};
sparsetype = CSC_MAT;
indexvals = {1, 3, 0, 2, 0, 1, 3, 4};
indexptrs = {0, 2, 4, 5, 8};
```

or

```
M = 5;
N = 4;
NNZ = 10;
NP = N;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0, *, *};
sparsetype = CSC_MAT;
indexvals = {1, 3, 0, 2, 0, 1, 3, 4, *, *};
indexptrs = {0, 2, 4, 5, 8};
```

where the first has no unused space, and the second has additional storage (the entries marked with * may contain any values). Note in both cases that the final value in `indexptrs` is 8, indicating the total number of nonzero entries in the matrix.

Similarly, in CSR format, the same matrix could be stored as

```
M = 5;
N = 4;
NNZ = 8;
NP = N;
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.

The following macros are provided to access the content of a `SUNMATRIX_SPARSE` matrix. The prefix `SM_` in the names denotes that these macros are for *SUNMatrix* implementations, and the suffix `_S` denotes that these are specific to the *sparse* version.

- `SM_CONTENT_S`

This routine gives access to the contents of the sparse `SUNMatrix`.

The assignment `A_cont = SM_CONTENT_S(A)` sets `A_cont` to be a pointer to the sparse `SUNMatrix` content structure.

Implementation:

```
#define SM_CONTENT_S(A)      ( (SUNMatrixContent_Sparse)(A->content) )
```

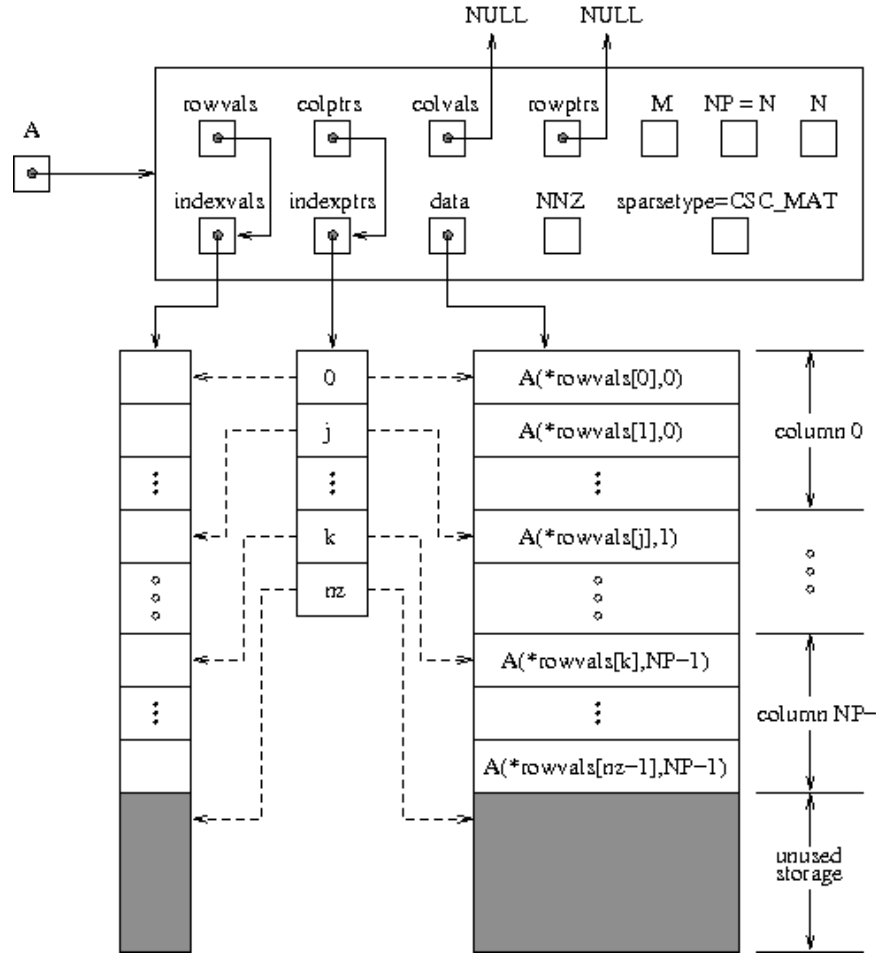


Figure 7.2: Diagram of the storage for a compressed-sparse-column matrix. Here A is an $M \times N$ sparse matrix with storage for up to NNZ nonzero entries (the allocated length of both `data` and `indexvals`). The entries in `indexvals` may assume values from 0 to $M - 1$, corresponding to the row index (zero-based) of each nonzero value. The entries in `data` contain the values of the nonzero entries, with the row i , column j entry of A (again, zero-based) denoted as $A(i, j)$. The `indexptrs` array contains $N + 1$ entries; the first N denote the starting index of each column within the `indexvals` and `data` arrays, while the final entry points one past the final nonzero entry. Here, although NNZ values are allocated, only `nz` are actually filled in; the greyed-out portions of `data` and `indexvals` indicate extra allocated space.

- `SM_ROWS_S`, `SM_COLUMNS_S`, `SM_NNZ_S`, `SM_NP_S`, and `SM_SPARSETYPE_S`

These macros give individual access to various lengths relevant to the content of a sparse SUNMatrix.

These may be used either to retrieve or to set these values. For example, the assignment `A_rows = SM_ROWS_S(A)` sets `A_rows` to be the number of rows in the matrix `A`. Similarly, the assignment `SM_COLUMNS_S(A) = A_cols` sets the number of columns in `A` to equal `A_cols`.

Implementation:

```
#define SM_ROWS_S(A)          ( SM_CONTENT_S(A)->M )
#define SM_COLUMNS_S(A)      ( SM_CONTENT_S(A)->N )
#define SM_NNZ_S(A)          ( SM_CONTENT_S(A)->NNZ )
#define SM_NP_S(A)           ( SM_CONTENT_S(A)->NP )
#define SM_SPARSETYPE_S(A)   ( SM_CONTENT_S(A)->sparsetype )
```

- `SM_DATA_S`, `SM_INDEXVALS_S`, and `SM_INDEXPTRS_S`

These macros give access to the `data` and index arrays for the matrix entries.

The assignment `A_data = SM_DATA_S(A)` sets `A_data` to be a pointer to the first component of the data array for the sparse SUNMatrix `A`. The assignment `SM_DATA_S(A) = A_data` sets the data array of `A` to be `A_data` by storing the pointer `A_data`.

Similarly, the assignment `A_indexvals = SM_INDEXVALS_S(A)` sets `A_indexvals` to be a pointer to the array of index values (i.e. row indices for a CSC matrix, or column indices for a CSR matrix) for the sparse SUNMatrix `A`. The assignment `A_indexptrs = SM_INDEXPTRS_S(A)` sets `A_indexptrs` to be a pointer to the array of index pointers (i.e. the starting indices in the data/indexvals arrays for each row or column in CSR or CSC formats, respectively).

Implementation:

```
#define SM_DATA_S(A)          ( SM_CONTENT_S(A)->data )
#define SM_INDEXVALS_S(A)     ( SM_CONTENT_S(A)->indexvals )
#define SM_INDEXPTRS_S(A)     ( SM_CONTENT_S(A)->indexptrs )
```

The `SUNMATRIX_SPARSE` module defines sparse implementations of all matrix operations listed in Table 7.2. Their names are obtained from those in Table 7.2 by appending the suffix `_Sparse` (e.g. `SUNMatCopy_Sparse`). The module `SUNMATRIX_SPARSE` provides the following additional user-callable routines:

- `SUNSparseMatrix`

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`).

```
SUNMatrix SUNSparseMatrix(sunindextype M, sunindextype N,
                          sunindextype NNZ, int sparsetype);
```

- `SUNSparseFromDenseMatrix`

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.

```
SUNMatrix SUNSparseFromDenseMatrix(SUNMatrix A, realtype droptol,
                                   int sparsetype);
```

- **SUNSparseFromBandMatrix**

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.

```
SUNMatrix SUNSparseFromBandMatrix(SUNMatrix A, realtype droptol,
                                   int sparsetype);
```

- **SUNSparseMatrix_Realloc**

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).

```
int SUNSparseMatrix_Realloc(SUNMatrix A);
```

- **SUNSparseMatrix_Reallocate**

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).

```
int SUNSparseMatrix_Reallocate(SUNMatrix A, sunindextype NNZ);
```

- **SUNSparseMatrix_Print**

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.

```
void SUNSparseMatrix_Print(SUNMatrix A, FILE* outfile);
```

- **SUNSparseMatrix_Rows**

This function returns the number of rows in the sparse `SUNMatrix`.

```
sunindextype SUNSparseMatrix_Rows(SUNMatrix A);
```

- **SUNSparseMatrix_Columns**

This function returns the number of columns in the sparse `SUNMatrix`.

```
sunindextype SUNSparseMatrix_Columns(SUNMatrix A);
```

- `SUNSparseMatrix_NNZ`

This function returns the number of entries allocated for nonzero storage for the sparse matrix `SUNMatrix`.

```
sunindextype SUNSparseMatrix_NNZ(SUNMatrix A);
```

- `SUNSparseMatrix_NP`

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.

```
sunindextype SUNSparseMatrix_NP(SUNMatrix A);
```

- `SUNSparseMatrix_SparseType`

This function returns the storage type (`CSR_MAT` or `CSC_MAT`) for the sparse `SUNMatrix`.

```
int SUNSparseMatrix_SparseType(SUNMatrix A);
```

- `SUNSparseMatrix_Data`

This function returns a pointer to the data array for the sparse `SUNMatrix`.

```
realtype* SUNSparseMatrix_Data(SUNMatrix A);
```

- `SUNSparseMatrix_IndexValues`

This function returns a pointer to index value array for the sparse `SUNMatrix`: for CSR format this is the column index for each nonzero entry, for CSC format this is the row index for each nonzero entry.

```
sunindextype* SUNSparseMatrix_IndexValues(SUNMatrix A);
```

- `SUNSparseMatrix_IndexPointers`

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.

```
sunindextype* SUNSparseMatrix_IndexPointers(SUNMatrix A);
```

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`, and `NVECTOR_PTHREADS`. As additional compatible vector implementations are added to `SUNDIALS`, these will be included within this compatibility check.

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.



7.4 SUNMatrix Examples

There are `SUNMatrix` examples that may be installed for each implementation: dense, banded, and sparse. Each implementation makes use of the functions in `test_sunmatrix.c`. These example functions show simple usage of the `SUNMatrix` family of functions. The inputs to the examples depend on the matrix type, and are output to `stdout` if the example is run without the appropriate number of command-line arguments.

The following is a list of the example functions in `test_sunmatrix.c`:

- **Test_SUNMatGetID**: Verifies the returned matrix ID against the value that should be returned.
- **Test_SUNMatClone**: Creates clone of an existing matrix, copies the data, and checks that their values match.
- **Test_SUNMatZero**: Zeros out an existing matrix and checks that each entry equals 0.0.
- **Test_SUNMatCopy**: Clones an input matrix, copies its data to a clone, and verifies that all values match.
- **Test_SUNMatScaleAdd**: Given an input matrix A and an input identity matrix I , this test clones and copies A to a new matrix B , computes $B = -B + B$, and verifies that the resulting matrix entries equal 0.0. Additionally, if the matrix is square, this test clones and copies A to a new matrix D , clones and copies I to a new matrix C , computes $D = D + I$ and $C = C + A$ using **SUNMatScaleAdd**, and then verifies that $C == D$.
- **Test_SUNMatScaleAddI**: Given an input matrix A and an input identity matrix I , this clones and copies I to a new matrix B , computes $B = -B + I$ using **SUNMatScaleAddI**, and verifies that the resulting matrix entries equal 0.0.
- **Test_SUNMatMatvec**: Given an input matrix A and input vectors x and y such that $y = Ax$, this test has different behavior depending on whether A is square. If it is square, it clones and copies A to a new matrix B , computes $B = 3B + I$ using **SUNMatScaleAddI**, clones y to new vectors w and z , computes $z = Bx$ using **SUNMatMatvec**, computes $w = 3y + x$ using **N_VLinearSum**, and verifies that $w == z$. If A is not square, it just clones y to a new vector z , computes $z = Ax$ using **SUNMatMatvec**, and verifies that $y == z$.
- **Test_SUNMatSpace** verifies that **SUNMatSpace** can be called, and outputs the results to **stdout**.

7.5 SUNMatrix functions used by CVODE

In Table 7.4, 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.

Table 7.4: List of matrix functions usage by CVODE code modules

	CVLS	CVBANDPRE	CVBBDPRE
SUNMatGetID	✓		
SUNMatClone	✓		
SUNMatDestroy	✓	✓	✓
SUNMatZero	✓	✓	✓
SUNMatCopy	✓	✓	✓
SUNMatScaleAddI	✓	✓	✓
SUNMatSpace	†	†	†

The matrix functions listed in Table 7.2 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 Table 7.2 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.

Chapter 8

Description of the SUNLinearSolver module

For problems that involve the solution of linear systems of equations, the SUNDIALS packages operate using generic linear solver modules defined through the SUNLINSOL API. This allows SUNDIALS packages to utilize any valid SUNLINSOL implementation that provides a set of required functions. These functions can be divided into three categories. The first are the core linear solver functions. The second group consists of “set” routines to supply the linear solver object with functions provided by the SUNDIALS package, or for modification of solver parameters. The last group consists of “get” routines for retrieving artifacts (statistics, residual vectors, etc.) from the linear solver. All of these functions are defined in the header file `sundials/sundials_linearsolver.h`.

The implementations provided with SUNDIALS work in coordination with the SUNDIALS generic NVECTOR and SUNMATRIX modules to provide a set of compatible data structures and solvers for the solution of linear systems using direct or matrix-free iterative 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 supported.

The iterative linear solvers packaged with SUNDIALS leverage scaling and preconditioning, as applicable, to balance error between solution components and to accelerate convergence of the linear solver. To this end, instead of solving the linear system $Ax = b$ directly, these apply the underlying iterative algorithm to the transformed system

$$\tilde{A}\tilde{x} = \tilde{b} \tag{8.1}$$

where

$$\begin{aligned} \tilde{A} &= S_1 P_1^{-1} A P_2^{-1} S_2^{-1}, \\ \tilde{b} &= S_1 P_1^{-1} b, \\ \tilde{x} &= S_2 P_2 x, \end{aligned} \tag{8.2}$$

and where

- P_1 is the left preconditioner,
- P_2 is the right preconditioner,
- S_1 is a diagonal matrix of scale factors for $P_1^{-1}b$,
- S_2 is a diagonal matrix of scale factors for P_2x .

SUNDIALS packages request that iterative linear solvers stop based on the 2-norm of the scaled preconditioned residual meeting a prescribed tolerance

$$\|\tilde{b} - \tilde{A}\tilde{x}\|_2 < \text{tol}.$$

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. 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 SUNLINSOL implementation, or for each SUNDIALS package.

8.0.1 SUNLinearSolver core functions

The core linear solver functions consist of four required routines to get the linear solver type (`SUNLinSolGetType`), initialize the linear solver object once all solver-specific options have been set (`SUNLinSolInitialize`), set up the linear solver object to utilize an updated matrix A (`SUNLinSolSetup`), and solve the linear system $Ax = b$ (`SUNLinSolSolve`). The remaining routine for destruction of the linear solver object (`SUNLinSolFree`) is 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 or iterative) from the abstract `SUNLinearSolver` interface. This is used to assess compatibility with SUNDIALS-provided linear solver interfaces.

Arguments LS (`SUNLinearSolver`) a SUNLINSOL object.

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

`SUNNONLINEARSOLVER_DIRECT` 0, the SUNLINSOL module uses direct methods to solve the linear system.

`SUNNONLINEARSOLVER_ITERATIVE` 1, the SUNLINSOL module iteratively solves the linear system, stopping when the linear residual is within a prescribed tolerance.

Notes

`SUNLinSolInitialize`

Call `retval = SUNLinSolInitialize(LS);`

Description The *required* function `SUNLinSolInitialize` performs linear solver initialization (assumes 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 8.1.

Notes

SUNLinSolSetup

Call	<code>retval = SUNLinSolSetup(LS, A);</code>
Description	The <i>required</i> function <code>SUNLinSolSetup</code> performs any linear solver setup needed, based on an updated system SUNMATRIX <code>A</code> . 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	<code>LS</code> (<code>SUNLinearSolver</code>) a SUNLINSOL object. <code>A</code> (<code>SUNMatrix</code>) 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 8.1.
Notes	

SUNLinSolSolve

Call	<code>retval = SUNLinSolSolve(LS, A, x, b, tol);</code>
Description	The <i>required</i> function <code>SUNLinSolSolve</code> solves a linear system $Ax = b$.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) a SUNLINSOL object. <code>A</code> (<code>SUNMatrix</code>) a SUNMATRIX object. <code>x</code> (<code>N_Vector</code>) a NVECTOR object. <code>b</code> (<code>N_Vector</code>) a NVECTOR object. <code>tol</code> (<code>realtype</code>) 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 8.1.
Notes	Direct solvers: can ignore the <code>*tol*</code> argument. Matrix-free solvers: can ignore the SUNMATRIX input <code>A</code> since a NULL argument will be passed (these should instead rely on the matrix-vector product function supplied through the routine <code>SUNLinSolSetATimes</code> . Iterative solvers: These should attempt to solve to the specified tolerance <code>tol</code> in a weighted 2-norm. If the solver does not support scaling then it should just use a 2-norm.

SUNLinSolFree

Call	<code>retval = SUNLinSolFree(LS);</code>
Description	The <i>optional</i> function <code>SUNLinSolFree</code> frees memory allocated by the linear solver.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) a SUNLINSOL object.
Return value	This should return zero for a successful call and a negative value for a failure.
Notes	

8.0.2 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	<code>retval = SUNLinSolSetATimes(LS, A_data, ATimes);</code>
Description	<p>The function <code>SUNLinSolSetATimes</code> is required for matrix-free linear solvers; otherwise it is optional.</p> <p>This routine provides an <code>ATimesFn</code> function pointer, as well as a <code>void *</code> 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.</p>
Arguments	<p><code>LS</code> (<code>SUNLinearSolver</code>) a SUNLINSOL object.</p> <p><code>A_data</code> (<code>void*</code>) data structure passed to <code>ATimes</code>.</p> <p><code>ATimes</code> (<code>ATimesFn</code>) function pointer implementing the matrix-vector product routine.</p>
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 8.1.
Notes	

SUNLinSolSetPreconditioner

Call	<code>retval = SUNLinSolSetPreconditioner(LS, Pdata, Pset, Psol);</code>
Description	<p>The <i>optional</i> function <code>SUNLinSolSetPreconditioner</code> provides <code>PSetupFn</code> and <code>PSolveFn</code> function pointers that implement the preconditioner solves P_1^{-1} and P_2^{-1} from equations (8.1)-(8.2). This routine will be called by a SUNDIALS package, which will provide translation between the generic <code>Pset</code> and <code>Psol</code> calls and the package- or user-supplied routines.</p>
Arguments	<p><code>LS</code> (<code>SUNLinearSolver</code>) a SUNLINSOL object.</p> <p><code>Pdata</code> (<code>void*</code>) data structure passed to both <code>Pset</code> and <code>Psol</code>.</p> <p><code>Pset</code> (<code>PSetupFn</code>) function pointer implementing the preconditioner setup.</p> <p><code>Psol</code> (<code>PSolveFn</code>) function pointer implementing the preconditioner solve.</p>
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 8.1.
Notes	

SUNLinSolSetScalingVectors

Call	<code>retval = SUNLinSolSetScalingVectors(LS, s1, s2);</code>
Description	<p>The <i>optional</i> function <code>SUNLinSolSetScalingVectors</code> provides left/right scaling vectors for the linear system solve. Here, <code>s1</code> and <code>s2</code> are <code>NVECTOR</code> of positive scale factors containing the diagonal of the matrices S_1 and S_2 from equations (8.1)-(8.2), respectively. Neither of these vectors need to be tested for positivity, and a <code>NULL</code> argument for either indicates that the corresponding scaling matrix is the identity.</p>
Arguments	<p><code>LS</code> (<code>SUNLinearSolver</code>) a SUNLINSOL object.</p> <p><code>s1</code> (<code>N_Vector</code>) diagonal of the matrix S_1</p> <p><code>s2</code> (<code>N_Vector</code>) diagonal of the matrix S_2</p>
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 8.1.
Notes	

8.0.3 SUNLinearSolver get functions

The following get functions allow SUNDIALS packages to retrieve results from the 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

Notes

SUNLinSolResNorm

Call `rnorm = SUNLinSolResNorm(LS);`

Description The *optional* function `SUNLinSolResNorm` should return the final residual norm from the last ‘solve’ call.

Arguments `LS` (`SUNLinearSolver`) a `SUNLINSOL` object.

Return value `realtype` containing the final residual norm

Notes

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 left `NULL` in the implementation.

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 `long int` containing the most recent error flag

Notes

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.

	<code>lrw (long int*)</code> the number of realtype words stored by the linear solver.
	<code>liw (long int*)</code> 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 8.1.
Notes	This function is advisory only, for use in determining a user's total space requirements.

8.0.4 Functions provided by SUNDIALS packages

To interface with the SUNLINSOL modules, the SUNDIALS packages supply a variety of routines for evaluating the matrix-vector product, and setting up and applying the preconditioner. These package-provided routines translate between the user-supplied ODE, DAE or nonlinear systems and the generic interfaces to the linear systems of equations that result in their solution. The types for functions provided to a SUNLINSOL module are defined in the header file `sundials/sundials_iterative.h`, and are described below.

ATimesFn

Definition	<code>typedef int (*ATimesFn)(void *A_data, N_Vector v, N_Vector z);</code>
Purpose	These functions compute the action of a matrix on a vector, performing the operation $z = Av$. Memory for <code>z</code> should already be allocated prior to calling this function. The vector <code>v</code> should be left unchanged.
Arguments	<code>A_data</code> is a pointer to client data, the same as that supplied to <code>SUNLinSolSetATimes</code> . <code>v</code> is the input vector to multiply. <code>z</code> is the output vector computed.
Return value	This routine should return 0 if successful and a non-zero value if unsuccessful.
Notes	

PSetupFn

Definition	<code>typedef int (*PSetupFn)(void *P_data)</code>
Purpose	These functions set up any requisite problem data in preparation for calls to the corresponding <code>PSolveFn</code> .
Arguments	<code>P_data</code> is a pointer to client data, the same pointer as that supplied to the routine <code>SUNLinSolSetPreconditioner</code> .
Return value	This routine should return 0 if successful and a non-zero value if unsuccessful.
Notes	

PSolveFn

Definition	<code>typedef int (*PSolveFn)(void *P_data, N_Vector r, N_Vector z, realtype tol, int lr)</code>
Purpose	These functions solve the preconditioner equation $Pz = r$ for the vector z . Memory for <code>z</code> should already be allocated prior to calling this function. The parameter <code>P_data</code> is a pointer to any information about P which the function needs in order to do its job (set up by the corresponding <code>PSetupFn</code>). The parameter <code>lr</code> is input, and indicates whether P is to be taken as the left preconditioner or the right preconditioner: <code>lr = 1</code> for left and <code>lr = 2</code> for right. If preconditioning is on one side only, <code>lr</code> can be ignored. If the preconditioner is iterative, then it should strive to solve the preconditioner equation so that

$$\|Pz - r\|_{\text{wrms}} < \text{tol}$$

where the weight vector for the WRMS norm may be accessed from the main package memory structure. The vector **r** should not be modified by the **PSolveFn**.

Arguments **P_data** is a pointer to client data, the same pointer as that supplied to the routine **SUNLinSolSetPreconditioner**.

r is the right-hand side vector for the preconditioner system

z is the solution vector for the preconditioner system

tol is the desired tolerance for an iterative preconditioner

lr is flag indicating whether the routine should perform left (1) or right (2) preconditioning.

Return value This routine should return 0 if successful and a non-zero value if unsuccessful. On a failure, a negative return value indicates an unrecoverable condition, while a positive value indicates a recoverable one, in which the calling routine may reattempt the solution after updating preconditioner data.

Notes

8.0.5 SUNLinearSolver return codes

The functions provided to SUNLINSOL modules by each SUNDIALS package, and functions within the SUNDIALS-provided SUNLINSOL implementations utilize a common set of return codes, shown in the Table 8.1. These adhere to a common pattern: 0 indicates success, a positive value corresponds to a recoverable failure, and a negative value indicates a non-recoverable failure. Aside from this pattern, the actual values of each error code are primarily to provide additional information to the user in case of a linear solver failure.

Table 8.1: Description of the **SUNLinearSolver** error codes

Name	Value	Description
SUNLS_SUCCESS	0	successful call or converged solve
SUNLS_MEM_NULL	-1	the memory argument to the function is NULL
SUNLS_ILL_INPUT	-2	an illegal input has been provided to the function
SUNLS_MEM_FAIL	-3	failed memory access or allocation
SUNLS_ATIMES_FAIL_UNREC	-4	an unrecoverable failure occurred in the ATimes routine
SUNLS_PSET_FAIL_UNREC	-5	an unrecoverable failure occurred in the Pset routine
SUNLS_PSOLVE_FAIL_UNREC	-6	an unrecoverable failure occurred in the Psolve routine
SUNLS_PACKAGE_FAIL_UNREC	-7	an unrecoverable failure occurred in an external linear solver package
SUNLS_GS_FAIL	-8	a failure occurred during Gram-Schmidt orthogonalization (SUNLINSOL_SPGMR/SUNLINSOL_SPGFMR)
SUNLS_QRSOL_FAIL	-9	a singular <i>R</i> matrix was encountered in a QR factorization (SUNLINSOL_SPGMR/SUNLINSOL_SPGFMR)

continued on next page

Name	Value	Description
SUNLS_RES_REDUCED	1	an iterative solver reduced the residual, but did not converge to the desired tolerance
SUNLS_CONV_FAIL	2	an iterative solver did not converge (and the residual was not reduced)
SUNLS_ATIMES_FAIL_REC	3	a recoverable failure occurred in the <code>ATimes</code> routine
SUNLS_PSET_FAIL_REC	4	a recoverable failure occurred in the <code>Pset</code> routine
SUNLS_PSOLVE_FAIL_REC	5	a recoverable failure occurred in the <code>Psolve</code> routine
SUNLS_PACKAGE_FAIL_REC	6	a recoverable failure occurred in an external linear solver package
SUNLS_QRFACT_FAIL	7	a singular matrix was encountered during a QR factorization (SUNLINSOL_SPGMR/SUNLINSOL_SPGMR)
SUNLS_LUFACT_FAIL	8	a singular matrix was encountered during a LU factorization (SUNLINSOL_DENSE/SUNLINSOL_BAND)

8.0.6 The generic SUNLinearSolver module

SUNDIALS packages interact with specific SUNLINSOL implementations through the generic SUNLINSOL module on which all other SUNLINSOL implementations are built. The `SUNLinearSolver` type is a pointer to a structure containing an implementation-dependent *content* field, and an *ops* field. The type `SUNLinearSolver` is defined as

```
typedef struct _generic_SUNLinearSolver *SUNLinearSolver;
```

```
struct _generic_SUNLinearSolver {
    void *content;
    struct _generic_SUNLinearSolver_Ops *ops;
};
```

where the `_generic_SUNLinearSolver_Ops` structure is a list of pointers to the various actual linear solver operations provided by a specific implementation. The `_generic_SUNLinearSolver_Ops` structure is defined as

```
struct _generic_SUNLinearSolver_Ops {
    SUNLinearSolver_Type (*gettype)(SUNLinearSolver);
    int (*setatimes)(SUNLinearSolver, void*, ATimesFn);
    int (*setpreconditioner)(SUNLinearSolver, void*,
                             PSetupFn, PSolveFn);
    int (*setscalingvectors)(SUNLinearSolver,
                             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);
    long int (*lastflag)(SUNLinearSolver);
    int (*space)(SUNLinearSolver, long int*, long int*);
    N_Vector (*resid)(SUNLinearSolver);
    int (*free)(SUNLinearSolver);
};
```

The generic SUNLINSOL module defines and implements the linear solver operations defined in Sections 8.0.1-8.0.3. 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));
}
```

8.1 Compatibility of SUNLinearSolver modules

We note that not all SUNLINSOL types are compatible with all SUNMATRIX and NVECTOR types provided with SUNDIALS. In Table 8.2 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 8.2: SUNDIALS matrix-based linear solvers and matrix implementations that can be used for each.

Linear Solver Interface	Dense Matrix	Banded Matrix	Sparse Matrix	User Supplied
Dense	✓			✓
Band		✓		✓
LapackDense	✓			✓
LapackBand		✓		✓
KLU			✓	✓
SUPERLUMT			✓	✓
User supplied	✓	✓	✓	✓

8.2 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 documentation for each SUNDIALS linear solver interface to determine which SUNLINSOL operations they require. 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.

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.

8.3 The `SUNLinearSolver_Dense` implementation

The dense implementation of the `SUNLINSOL` module provided with `SUNDIALS`, `SUNLINSOL_DENSE`, 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`).

8.3.1 `SUNLINSOL_DENSE` usage

The header file to include when using this module is `sunlinsol/sunlinsol_dense.h`. The `SUNLINSOL_DENSE` module is accessible from all `SUNDIALS` solvers *without* linking to the `libsundials_sunlinsoldense` module library.

The module `SUNLINSOL_DENSE` provides the following user-callable constructor routine:

`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 consistent `NVECTOR` and `SUNMATRIX` implementations. These are currently limited to the `SUNMATRIX_DENSE` matrix type and the `NVECTOR_SERIAL`, `NVECTOR_OPENMP`, and `NVECTOR_PTHREADS` vector types. As additional compatible matrix and vector implementations are added to `SUNDIALS`, these will be included within this compatibility check.

For backwards compatibility, we also provide the wrapper function,

- `SUNDenseLinearSolver`

Wrapper function for `SUNLinSol_Dense`, with identical input and output arguments.

For solvers that include a Fortran 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

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.

8.3.2 SUNLINSOL_DENSE description

The SUNLINSOL_DENSE module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_Dense {
    sunindextype N;
    sunindextype *pivots;
    long int last_flag;
};
```

These entries of the *content* field contain the following information:

N - size of the linear system,

pivots - index array for partial pivoting in LU factorization,

last_flag - last error return flag from internal function evaluations.

This solver is constructed to perform the following operations:

- The “setup” call performs a *LU* factorization with partial (row) pivoting ($\mathcal{O}(N^3)$ cost), $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix with 1’s on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX_DENSE object A , with pivoting information encoding P stored in the **pivots** array.
- The “solve” call performs pivoting and forward and backward substitution using the stored **pivots** array and the *LU* factors held in the SUNMATRIX_DENSE object ($\mathcal{O}(N^2)$ cost).

The SUNLINSOL_DENSE module defines dense implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- 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

8.4 The SUNLinearSolver_Band implementation

The band implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_BAND, 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).

8.4.1 SUNLINSOL_BAND usage

The header file to include when using this module is `sunlinsol/sunlinsol.band.h`. The SUNLINSOL_BAND module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolband` module library.

The module SUNLINSOL_BAND provides the following user-callable constructor routine:

SUNLinSol_Band

Call	<code>LS = SUNLinSol_Band(y, A);</code>
Description	The function <code>SUNLinSol_Band</code> creates and allocates memory for a band <code>SUNLinearSolver</code> object.
Arguments	<code>y</code> (<code>N_Vector</code>) a template for cloning vectors needed within the solver <code>A</code> (<code>SUNMatrix</code>) a <code>SUNMATRIX_BAND</code> matrix template for cloning matrices needed within the solver
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>A</code> or <code>y</code> are incompatible then this routine will return <code>NULL</code> .
Notes	This routine will perform consistency checks to ensure that it is called with consistent <code>NVECTOR</code> and <code>SUNMATRIX</code> implementations. These are currently limited to the <code>SUNMATRIX_DENSE</code> matrix type and the <code>NVECTOR_SERIAL</code> , <code>NVECTOR_OPENMP</code> , and <code>NVECTOR_PTHREADS</code> vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check. Additionally, this routine will verify that the input matrix <code>A</code> is allocated with appropriate upper bandwidth storage for the <i>LU</i> factorization.

For backwards compatibility, we also provide the wrapper functions:

- `SUNBandLinearSolver`

Wrapper function for `SUNLinSol_Band`, with identical input and output arguments.

For solvers that include a Fortran interface module, the SUNLINSOL_BAND module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

FSUNBANDLINSOLINIT

Call	<code>FSUNBANDLINSOLINIT(code, ier)</code>
Description	The function <code>FSUNBANDLINSOLINIT</code> can be called for Fortran programs to create a band <code>SUNLinearSolver</code> object.
Arguments	<code>code</code> (<code>int*</code>) is an integer input specifying the solver id (1 for <code>CVODE</code> , 2 for <code>IDA</code> , 3 for <code>KINSOL</code> , and 4 for <code>ARKODE</code>).
Return value	<code>ier</code> is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	This routine must be called <i>after</i> both the <code>NVECTOR</code> and <code>SUNMATRIX</code> objects have been initialized.

Additionally, when using `ARKODE` with a non-identity mass matrix, the SUNLINSOL_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

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.

8.4.2 SUNLINSOL_BAND description

The SUNLINSOL_BAND module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_Band {
    sunindextype N;
    sunindextype *pivots;
    long int last_flag;
};
```

These entries of the *content* field contain the following information:

N - size of the linear system,

pivots - index array for partial pivoting in LU factorization,

last_flag - last error return flag from internal function evaluations.

This solver is constructed to perform the following operations:

- The “setup” call performs 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**.



The SUNLINSOL_BAND module defines band implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- SUNLinSolGetType_Band
- SUNLinSolInitialize_Band – this does nothing, since all consistency checks are performed at solver creation.
- SUNLinSolSetup_Band – this performs the *LU* factorization.
- SUNLinSolSolve_Band – this uses the *LU* factors and **pivots** array to perform the solve.
- SUNLinSolLastFlag_Band
- SUNLinSolSpace_Band – this only returns information for the storage *within* the solver object, i.e. storage for **N**, **last_flag**, and **pivots**.
- SUNLinSolFree_Band

8.5 The SUNLinearSolver_LapackDense implementation

The LAPACK dense implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_LAPACKDENSE, 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).

8.5.1 SUNLINSOL_LAPACKDENSE usage

The header file to include when using this module is `sunlinsol/sunlinsol_lapackdense.h`. The installed module library to link to is `libsundials_sunlinsollapackdense.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The module SUNLINSOL_LAPACKDENSE provides the following user-callable constructor routine:

SUNLinSol_LapackDense

Call	<code>LS = SUNLinSol_LapackDense(y, A);</code>
Description	The function <code>SUNLinSol_LapackDense</code> creates and allocates memory for a LAPACK-based, dense <code>SUNLinearSolver</code> object.
Arguments	<code>y</code> (<code>N_Vector</code>) a template for cloning vectors needed within the solver <code>A</code> (<code>SUNMatrix</code>) a <code>SUNMATRIX_DENSE</code> matrix template for cloning matrices needed within the solver
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>A</code> or <code>y</code> are incompatible then this routine will return <code>NULL</code> .
Notes	This routine will perform consistency checks to ensure that it is called with consistent <code>NVECTOR</code> and <code>SUNMATRIX</code> implementations. These are currently limited to the <code>SUNMATRIX_DENSE</code> matrix type and the <code>NVECTOR_SERIAL</code> , <code>NVECTOR_OPENMP</code> , and <code>NVECTOR_PTHREADS</code> vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

For backwards compatibility, we also provide the wrapper function,

- `SUNLapackDense`

Wrapper function for `SUNLinSol_LapackDense`, with identical input and output arguments.

For solvers that include a Fortran interface module, the SUNLINSOL_LAPACKDENSE module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

FSUNLAPACKDENSEINIT

Call	<code>FSUNLAPACKDENSEINIT(code, ier)</code>
Description	The function <code>FSUNLAPACKDENSEINIT</code> can be called for Fortran programs to create a LAPACK-based dense <code>SUNLinearSolver</code> object.
Arguments	<code>code</code> (<code>int*</code>) is an integer input specifying the solver id (1 for <code>CVODE</code> , 2 for <code>IDA</code> , 3 for <code>KINSOL</code> , and 4 for <code>ARKODE</code>).
Return value	<code>ier</code> is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	This routine must be called <i>after</i> both the <code>NVECTOR</code> and <code>SUNMATRIX</code> objects have been initialized.

Additionally, when using `ARKODE` with a non-identity mass matrix, the SUNLINSOL_LAPACKDENSE module includes a Fortran-callable function for creating a `SUNLinearSolver` mass matrix solver object.

FSUNMASSLAPACKDENSEINIT

Call `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

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.

8.5.2 SUNLINSOL_LAPACKDENSE description

The `SUNLINSOL_LAPACKDENSE` module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_Dense {
    sunindextype N;
    sunindextype *pivots;
    long int last_flag;
};
```

These entries of the *content* field contain the following information:

N - size of the linear system,

pivots - index array for partial pivoting in LU factorization,

last_flag - last error return flag from internal function evaluations.

The `SUNLINSOL_LAPACKDENSE` module is a `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 `int64_t` for the `sunindextype`.



This solver is constructed to perform the following operations:

- The “setup” call performs a *LU* factorization with partial (row) pivoting ($\mathcal{O}(N^3)$ cost), $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix with 1’s on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input `SUNMATRIX_DENSE` object A , with pivoting information encoding P stored in the `pivots` array.
- The “solve” call performs pivoting and forward and backward substitution using the stored `pivots` array and the *LU* factors held in the `SUNMATRIX_DENSE` object ($\mathcal{O}(N^2)$ cost).

The `SUNLINSOL_LAPACKDENSE` module defines dense implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- `SUNLinSolGetType_LapackDense`
- `SUNLinSolInitialize_LapackDense` – this does nothing, since all consistency checks are performed at solver creation.

- `SUNLinSolSetup_LapackDense` – this calls either `DGETRF` or `SGETRF` to perform the *LU* factorization.
- `SUNLinSolSolve_LapackDense` – this calls either `DGETRS` or `SGETRS` to use the *LU* factors and `pivots` array to perform the solve.
- `SUNLinSolLastFlag_LapackDense`
- `SUNLinSolSpace_LapackDense` – this only returns information for the storage *within* the solver object, i.e. storage for `N`, `last_flag`, and `pivots`.
- `SUNLinSolFree_LapackDense`

8.6 The SUNLinearSolver_LapackBand implementation

The LAPACK band implementation of the SUNLINSOL module provided with SUNDIALS, `SUNLINSOL_LAPACKBAND`, 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`).

8.6.1 SUNLINSOL_LAPACKBAND usage

The header file to include when using this module is `sunlinsol/sunlinsol_lapackband.h`. The installed module library to link to is `libsundials_sunlinsollapackband.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The module `SUNLINSOL_LAPACKBAND` provides the following user-callable routine:

SUNLinSol_LapackBand

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`.

Notes This routine will perform consistency checks to ensure that it is called with consistent `NVECTOR` and `SUNMATRIX` implementations. These are currently limited to the `SUNMATRIX_BAND` matrix type and the `NVECTOR_SERIAL`, `NVECTOR_OPENMP`, and `NVECTOR_PTHREADS` vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

Additionally, this routine will verify that the input matrix `A` is allocated with appropriate upper bandwidth storage for the *LU* factorization.

For backwards compatibility, we also provide the wrapper functions:

- `SUNLapackBand`

Wrapper function for `SUNLinSol_LapackBand`, with identical input and output arguments.

For solvers that include a Fortran interface module, the `SUNLINSOL_LAPACKBAND` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

FSUNLAPACKBANDINIT

Call	FSUNLAPACKBANDINIT(<i>code</i> , <i>ier</i>)
Description	The function FSUNLAPACKBANDINIT can be called for Fortran programs to create a LAPACK-based band SUNLinearSolver object.
Arguments	<i>code</i> (<i>int*</i>) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
Return value	<i>ier</i> is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	This routine must be called <i>after</i> both the NVECTOR and SUNMATRIX objects have been initialized.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL_LAPACKBAND module includes a Fortran-callable function for creating a SUNLinearSolver mass matrix solver object.

FSUNMASSLAPACKBANDINIT

Call	FSUNMASSLAPACKBANDINIT(<i>ier</i>)
Description	The function FSUNMASSLAPACKBANDINIT can be called for Fortran programs to create a LAPACK-based, band SUNLinearSolver object for mass matrix linear systems.
Arguments	
Return value	<i>ier</i> is a <i>int</i> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	This routine must be called <i>after</i> both the NVECTOR and SUNMATRIX mass-matrix objects have been initialized.

8.6.2 SUNLINSOL_LAPACKBAND description

The SUNLINSOL_LAPACKBAND module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_Band {
    sunindextype N;
    sunindextype *pivots;
    long int last_flag;
};
```

These entries of the *content* field contain the following information:

N - size of the linear system,


pivots - index array for partial pivoting in LU factorization,

last_flag - last error return flag from internal function evaluations.

The SUNLINSOL_LAPACKBAND module is a 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 *int64.t* for the *sunindextype*.

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`.

The `SUNLINSOL_LAPACKBAND` module defines band implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- `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`

8.7 The SUNLinearSolver_KLU implementation

The KLU implementation of the `SUNLINSOL` module provided with SUNDIALS, `SUNLINSOL_KLU`, 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`).

8.7.1 SUNLINSOL_KLU usage

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 module `SUNLINSOL_KLU` provides the following user-callable routines:

<code>SUNLinSol_KLU</code>	
Call	<code>LS = SUNLinSol_KLU(y, A);</code>
Description	The function <code>SUNLinSol_KLU</code> creates and allocates memory for a <code>SUNLINSOL_KLU</code> object.
Arguments	<code>y</code> (<code>N_Vector</code>) a template for cloning vectors needed within the solver <code>A</code> (<code>SUNMatrix</code>) a <code>SUNMATRIX_SPARSE</code> matrix template for cloning matrices needed within the solver
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>A</code> or <code>y</code> are incompatible then this routine will return <code>NULL</code> .

Notes This routine will perform consistency checks to ensure that it is called with consistent 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.

SUNLinSol_KLUReInit

Call `retval = SUNLinSol_KLUReInit(LS, A, nnz, reinit_type);`

Description The function `SUNLinSol_KLUReInit` reinitializes memory and flags for a new factorization (symbolic and numeric) to be conducted at the next solver setup call. This routine is useful in the cases where the number of nonzeros has changed or if the structure of the linear system has changed which would require a new symbolic (and numeric factorization).

Arguments

LS	(<code>SUNLinearSolver</code>) a template for cloning vectors needed within the solver
A	(<code>SUNMatrix</code>) a <code>SUNMATRIX_SPARSE</code> matrix template for cloning matrices needed within the solver
nnz	(<code>sunindextype</code>) the new number of nonzeros in the matrix
reinit_type	(<code>int</code>) flag governing the level of reinitialization. The allowed values are: <ul style="list-style-type: none"> • <code>SUNKLU_REINIT_FULL</code> – The Jacobian matrix will be destroyed and a new one will be allocated based on the <code>nnz</code> value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup. • <code>SUNKLU_REINIT_PARTIAL</code> – Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of <code>nnz</code> given in the sparse matrix provided to the original constructor routine (or the previous <code>SUNLinSol_KLUReInit</code> call).

Return value The return values from this function are `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 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.

This routine assumes no other changes to solver use are necessary.

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	(<code>SUNLinearSolver</code>) the <code>SUNLINSOL_KLU</code> object
ordering	(<code>int</code>) flag indication the reordering algorithm to use. Options include: <ul style="list-style-type: none"> 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`), or `SUNLS_SUCCESS`.

Notes

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

- `SUNKLU`
Wrapper function for `SUNLinSol_KLU`
- `SUNKLUREInit`
Wrapper function for `SUNLinSol_KLUREInit`
- `SUNKLUSetOrdering`
Wrapper function for `SUNLinSol_KLUSetOrdering`

For solvers that include a Fortran 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 `SUNLINSOL_KLU` object.

Arguments `code` (`int*`) is an integer input specifying the solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, and 4 for `ARKODE`).

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_KLU` module includes a Fortran-callable function for creating a `SUNLinearSolver` mass matrix solver object.

FSUNMASSKLUIT

Call `FSUNMASSKLUIT(ier)`

Description The function `FSUNMASSKLUIT` can be called for Fortran programs to create a `SUNLINSOL_KLU` object for mass matrix linear systems.

Arguments

Return value `ier` is a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* both the `NVECTOR` and `SUNMATRIX` mass-matrix objects have been initialized.

The `SUNLinSol_KLUREInit` and `SUNLinSol_KLUSetOrdering` routines also support Fortran interfaces for the system and mass matrix solvers:

FSUNKLUREINIT

Call `FSUNKLUREINIT(code, nnz, reinit_type, ier)`

Description The function `FSUNKLUREINIT` can be called for Fortran programs to re-initialize a `SUNLINSOL_KLU` object.

Arguments	code	(int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
	nnz	(sunindextype*) the new number of nonzeros in the matrix
	reinit_type	(int*) flag governing the level of reinitialization. The allowed values are: <ul style="list-style-type: none"> 1 – The Jacobian matrix will be destroyed and a new one will be allocated based on the nnz value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup. 2 – Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of nnz given in the sparse matrix provided to the original constructor routine (or the previous SUNLinSol_KLUReInit call).
Return value	ier	is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes		See SUNLinSol_KLUReInit for complete further documentation of this routine.

FSUNMASSKLUREINIT

Call	FSUNMASSKLUREINIT (nnz , reinit_type , ier)	
Description	The function FSUNMASSKLUREINIT can be called for Fortran programs to re-initialize a SUNLINSOL_KLU object for mass matrix linear systems.	
Arguments	The arguments are identical to FSUNKLUREINIT above, except that code is not needed since mass matrix linear systems only arise in ARKODE.	
Return value	ier	is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_KLUReInit for complete further documentation of this routine.	

FSUNKLUSETORDERING

Call	FSUNKLUSETORDERING (code , ordering , ier)	
Description	The function FSUNKLUSETORDERING can be called for Fortran programs to change the reordering algorithm used by KLU.	
Arguments	code	(int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
	ordering	(int*) flag indication the reordering algorithm to use. Options include: <ul style="list-style-type: none"> 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.

8.7.2 SUNLINSOL_KLU description

The `SUNLINSOL_KLU` module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_KLU {
    long int      last_flag;
    int           first_factorize;
    sun_klu_symbolic *symbolic;
    sun_klu_numeric *numeric;
    sun_klu_common common;
    sunindextype  (*klu_solver)(sun_klu_symbolic*, sun_klu_numeric*,
                                sunindextype, sunindextype,
                                double*, sun_klu_common*);
};
```

These entries of the *content* field contain the following information:

last_flag - last error return flag from internal function evaluations,

first_factorize - flag indicating whether the factorization has ever been performed,

symbolic - KLU storage structure for symbolic factorization components,

numeric - KLU storage structure for numeric factorization components,

common - storage structure for common KLU solver components,

klu_solver – pointer to the appropriate KLU solver function (depending on whether it is using a CSR or CSC sparse matrix).



The `SUNLINSOL_KLU` module is a `SUNLINSOL` wrapper for the KLU sparse matrix factorization and solver library written by Tim Davis [1, 14]. 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.

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.

The `SUNLINSOL_KLU` module defines implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- `SUNLinSolGetType_KLU`
- `SUNLinSolInitialize_KLU` – this sets the `first_factorize` flag to 1, forcing both symbolic and numerical factorizations on the subsequent “setup” call.
- `SUNLinSolSetup_KLU` – this performs either a *LU* factorization or refactorization of the input matrix.
- `SUNLinSolSolve_KLU` – this calls the appropriate KLU solve routine to utilize the *LU* factors to solve the linear system.
- `SUNLinSolLastFlag_KLU`
- `SUNLinSolSpace_KLU` – this only returns information for the storage within the solver *interface*, i.e. storage for the integers `last_flag` and `first_factorize`. For additional space requirements, see the KLU documentation.
- `SUNLinSolFree_KLU`

8.8 The SUNLinearSolver_SuperLUMT implementation

The SUPERLUMT implementation of the SUNLINSOL module provided with SUNDIALS, `SUNLINSOL_SUPERLUMT`, 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.

8.8.1 SUNLINSOL_SUPERLUMT usage

The header file to include when using this module is `sunlinsol/sunlinsol_superluml.h`. The installed module library to link to is `libsundials_sunlinsolsuperluml.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The module `SUNLINSOL_SUPERLUMT` provides the following user-callable routines:

SUNLinSol_SuperLUMT

Call	<code>LS = SUNLinSol_SuperLUMT(y, A, num_threads);</code>		
Description	The function <code>SUNLinSol_SuperLUMT</code> creates and allocates memory for a <code>SUNLINSOL_SUPERLUMT</code> object.		
Arguments	<code>y</code>	(N_Vector) a template for cloning vectors needed within the solver	
	<code>A</code>	(SUNMatrix) a <code>SUNMATRIX_SPARSE</code> matrix template for cloning matrices needed within the solver	
	<code>num_threads</code>	(int) desired number of threads (OpenMP or Pthreads, depending on how <code>SUPERLUMT</code> was installed) to use during the factorization steps	
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>A</code> or <code>y</code> are incompatible then this routine will return <code>NULL</code> .		
Notes	This routine analyzes the input matrix and vector to determine the linear system size and to assess compatibility with the <code>SUPERLUMT</code> library.		
	This routine will perform consistency checks to ensure that it is called with consistent <code>NVECTOR</code> and <code>SUNMATRIX</code> implementations. These are currently limited to the <code>SUNMATRIX_SPARSE</code> matrix type (using either CSR or CSC storage formats) and the <code>NVECTOR_SERIAL</code> , <code>NVECTOR_OPENMP</code> , and <code>NVECTOR_PTHREADS</code> vector types. As additional compatible matrix and vector implementations are added to <code>SUNDIALS</code> , these will be included within this compatibility check.		
	The <code>num_threads</code> argument is not checked and is passed directly to <code>SUPERLUMT</code> routines.		

SUNLinSol_SuperLUMTSetOrdering

Call	<code>retval = SUNLinSol_SuperLUMTSetOrdering(LS, ordering);</code>
Description	The function <code>SUNLinSol_SuperLUMTSetOrdering</code> sets the ordering used by <code>SUPERLUMT</code> for reducing fill in the linear solve.
Arguments	<code>LS</code> (SUNLinearSolver) the <code>SUNLINSOL_SUPERLUMT</code> object <code>ordering</code> (int) a flag indicating the ordering algorithm, options are: 0 natural ordering 1 minimal degree ordering on $A^T A$ 2 minimal degree ordering on $A^T + A$ 3 COLAMD ordering for unsymmetric matrices The default is 3 for COLAMD.
Return value	The return values from this function are <code>SUNLS_MEM_NULL</code> (<code>S</code> is <code>NULL</code>), <code>SUNLS_ILL_INPUT</code> (invalid <code>ordering_choice</code>), or <code>SUNLS_SUCCESS</code> .

Notes

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

- **SUNSuperLUMT**
Wrapper function for `SUNLinSol_SuperLUMT`
- **SUNSuperLUMTSetOrdering**
Wrapper function for `SUNLinSol_SuperLUMTSetOrdering`

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(<i>code</i> , <i>num_threads</i> , <i>ier</i>)
Description	The function FSUNSUPERLUMTINIT can be called for Fortran programs to create a SUNLINSOL_KLU object.
Arguments	<p><i>code</i> (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).</p> <p><i>num_threads</i> (int*) desired number of threads (OpenMP or Pthreads, depending on how SUPERLUMT was installed) to use during the factorization steps</p>
Return value	<i>ier</i> is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	This routine must be called <i>after</i> both the NVECTOR and SUNMATRIX objects have been initialized.

Additionally, when using ARKODE with a non-identity mass matrix, the SUNLINSOL_SUPERLUMT module includes a Fortran-callable function for creating a **SUNLinearSolver** mass matrix solver object.

FSUNMASSSUPERLUMTINIT

Call	FSUNMASSSUPERLUMTINIT(<i>num_threads</i> , <i>ier</i>)
Description	The function FSUNMASSSUPERLUMTINIT can be called for Fortran programs to create a SUNLINSOL_SUPERLUMT object for mass matrix linear systems.
Arguments	<i>num_threads</i> (int*) desired number of threads (OpenMP or Pthreads, depending on how SUPERLUMT was installed) to use during the factorization steps.
Return value	<i>ier</i> 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 <i>after</i> 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(<i>code</i> , <i>ordering</i> , <i>ier</i>)
Description	The function FSUNSUPERLUMTSETORDERING can be called for Fortran programs to update the ordering algorithm in a SUNLINSOL_SUPERLUMT object.
Arguments	<p><i>code</i> (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).</p> <p><i>ordering</i> (int*) a flag indicating the ordering algorithm, options are:</p> <ul style="list-style-type: none"> 0 natural ordering 1 minimal degree ordering on $A^T A$ 2 minimal degree ordering on $A^T + A$ 3 COLAMD ordering for unsymmetric matrices <p>The default is 3 for COLAMD.</p>
Return value	<i>ier</i> is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SuperLUMTSetOrdering for complete further documentation of this routine.

FSUNMASSUPERLUMTSETORDERING

Call	FSUNMASSUPERLUMTSETORDERING(ordering, ier)
Description	The function FSUNMASSUPERLUMTSETORDERING can be called for Fortran programs to update the ordering algorithm in a SUNLINSOL_SUPERLUMT object for mass matrix linear systems.
Arguments	<p>ordering (int*) a flag indicating the ordering algorithm, options are:</p> <ul style="list-style-type: none"> 0 natural ordering 1 minimal degree ordering on $A^T A$ 2 minimal degree ordering on $A^T + A$ 3 COLAMD ordering for unsymmetric matrices <p>The default is 3 for COLAMD.</p>
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.

8.8.2 SUNLINSOL_SUPERLUMT description

The SUNLINSOL_SUPERLUMT module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_SuperLUMT {
    long 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;
    superluml_options_t *options;
};
```

These entries of the *content* field contain the following information:

last_flag - last error return flag from internal function evaluations,

first_factorize - flag indicating whether the factorization has ever been performed,

A, AC, L, U, B - SuperMatrix pointers used in solve,

Gstat - GStat_t object used in solve,

perm_r, perm_c - permutation arrays used in solve,

N - size of the linear system,

num_threads - number of OpenMP/Pthreads threads to use,

diag_pivot_thresh - threshold on diagonal pivoting,

ordering - flag for which reordering algorithm to use,

options - pointer to SUPERLUMT options structure.



The SUNLINSOL_SUPERLUMT module is a SUNLINSOL wrapper for the SUPERLUMT sparse matrix factorization and solver library written by X. Sherry Li [2, 31, 16]. 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.

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.

The SUNLINSOL_SUPERLUMT module defines implementations of all “direct” linear solver operations listed in Sections 8.0.1-8.0.3:

- `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`

8.9 The SUNLinearSolver_SPGMR implementation

The SPGMR (Scaled, Preconditioned, Generalized Minimum Residual [36]) implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_SPGMR, is an iterative linear solver that is designed to be compatible with any NVECTOR implementation (serial, threaded, parallel, and user-supplied) 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.

8.9.1 SUNLINSOL_SPGMR usage

The header file to include when using this module is `sunlinsol/sunlinsol_spgmr.h`. The SUNLINSOL_SPGMR module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolspgmr` module library.

The module SUNLINSOL_SPGMR provides the following user-callable routines:

SUNLinSol_SPGMR

Call	<code>LS = SUNLinSol_SPGMR(y, pretype, maxl);</code>
Description	The function <code>SUNLinSol_SPGMR</code> creates and allocates memory for a SPGMR <code>SUNLinearSolver</code> .
Arguments	<p><code>y</code> (N_Vector) a template for cloning vectors needed within the solver</p> <p><code>pretype</code> (int) flag indicating the desired type of preconditioning, allowed values are:</p> <ul style="list-style-type: none"> • <code>PREC_NONE</code> (0) • <code>PREC_LEFT</code> (1) • <code>PREC_RIGHT</code> (2) • <code>PREC_BOTH</code> (3) <p>Any other integer input will result in the default (no preconditioning).</p> <p><code>maxl</code> (int) the number of Krylov basis vectors to use. values ≤ 0 will result in the default value (5).</p>
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>y</code> is incompatible then this routine will return <code>NULL</code> .
Notes	<p>This routine will perform consistency checks to ensure that it is called with a consistent NVECTOR implementation (i.e. that it supplies the requisite vector operations). If <code>y</code> is incompatible, then this routine will return <code>NULL</code>.</p> <p>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.</p>

SUNLinSol_SPGMRSetPrecType

Call	<code>retval = SUNLinSol_SPGMRSetPrecType(LS, pretype);</code>
Description	The function <code>SUNLinSol_SPGMRSetPrecType</code> updates the type of preconditioning to use in the SUNLINSOL_SPGMR object.
Arguments	<p><code>LS</code> (SUNLinearSolver) the SUNLINSOL_SPGMR object to update</p> <p><code>pretype</code> (int) flag indicating the desired type of preconditioning, allowed values match those discussed in <code>SUNLinSol_SPGMR</code>.</p>
Return value	This routine will return with one of the error codes <code>SUNLS_ILL_INPUT</code> (illegal <code>pretype</code>), <code>SUNLS_MEM_NULL</code> (S is NULL) or <code>SUNLS_SUCCESS</code> .
Notes	

SUNLinSol_SPGMRSetGSType

Call	<code>retval = SUNLinSol_SPGMRSetGSType(LS, gstype);</code>
Description	The function <code>SUNLinSol_SPGMRSetGSType</code> sets the type of Gram-Schmidt orthogonalization to use in the SUNLINSOL_SPGMR object.
Arguments	<p><code>LS</code> (SUNLinearSolver) the SUNLINSOL_SPGMR object to update</p> <p><code>gstype</code> (int) flag indicating the desired orthogonalization algorithm; allowed values are:</p>

- 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`.

Notes

`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`.

Notes

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

- `SUNSPGMR`
 Wrapper function for `SUNLinSol_SPGMR`
- `SUNSPGMRSetPrecType`
 Wrapper function for `SUNLinSol_SPGMRSetPrecType`
- `SUNSPGMRSetGSType`
 Wrapper function for `SUNLinSol_SPGMRSetGSType`
- `SUNSPGMRSetMaxRestarts`
 Wrapper function for `SUNLinSol_SPGMRSetMaxRestarts`

For solvers that include a Fortran interface module, the `SUNLINSOL_SPGMR` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

`FSUNSPGMRINIT`

Call `FSUNSPGMRINIT(code, pretype, maxl, ier)`

Description The function `FSUNSPGMRINIT` can be called for Fortran programs to create a `SUNLINSOL_SPGMR` object.

Arguments `code` (`int*`) is an integer input specifying the solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, and 4 for `ARKODE`).
`pretype` (`int*`) flag indicating desired preconditioning type
`maxl` (`int*`) flag indicating Krylov subspace size

Return value `ier` is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* the `NVECTOR` object has been initialized.

Allowable values for `pretype` and `maxl` are the same as for the C function `SUNLinSol_SPGMR`.

Additionally, when using `ARKODE` with a non-identity mass matrix, the `SUNLINSOL_SPGMR` module includes a Fortran-callable function for creating a `SUNLinearSolver` mass matrix solver object.

FSUNMASSSPGMRINIT

Call FSUNMASSSPGMRINIT(*pretype*, *maxl*, *ier*)

Description The function FSUNMASSSPGMRINIT can be called for Fortran programs to create a SUNLINSOL_SPGMR object for mass matrix linear systems.

Arguments *pretype* (*int**) flag indicating desired preconditioning type
maxl (*int**) flag indicating Krylov subspace size

Return value *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* the NVECTOR object has been initialized.
 Allowable values for *pretype* and *maxl* are the same as for the C function SUNLinSol_SPGMR. The SUNLinSol_SPGMRSetPrecType, SUNLinSol_SPGMRSetGSType and SUNLinSol_SPGMRSetMaxRestarts routines also support Fortran interfaces for the system and mass matrix solvers

FSUNSPGMRSETGSTYPE

Call FSUNSPGMRSETGSTYPE(*code*, *gstype*, *ier*)

Description The function FSUNSPGMRSETGSTYPE can be called for Fortran programs to change the Gram-Schmidt orthogonalization algorithm.

Arguments *code* (*int**) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
gstype (*int**) flag indicating the desired orthogonalization algorithm.

Return value *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol_SPGMRSetGSType for complete further documentation of this routine.

FSUNMASSSPGMRSETGSTYPE

Call FSUNMASSSPGMRSETGSTYPE(*gstype*, *ier*)

Description The function FSUNMASSSPGMRSETGSTYPE can be called for Fortran programs to change the Gram-Schmidt orthogonalization algorithm for mass matrix linear systems.

Arguments The arguments are identical to FSUNSPGMRSETGSTYPE above, except that *code* is not needed since mass matrix linear systems only arise in ARKODE.

Return value *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol_SPGMRSetGSType for complete further documentation of this routine.

FSUNSPGMRSETPRECTYPE

Call FSUNSPGMRSETPRECTYPE(*code*, *pretype*, *ier*)

Description The function FSUNSPGMRSETPRECTYPE can be called for Fortran programs to change the type of preconditioning to use.

Arguments *code* (*int**) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
pretype (*int**) flag indicating the type of preconditioning to use.

Return value *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol_SPGMRSetPrecType for complete further documentation of this routine.

FSUNMASSSPGMRSETPRECTYPE

Call	FSUNMASSSPGMRSETPRECTYPE(<i>pretype</i> , <i>ier</i>)
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 <i>code</i> is not needed since mass matrix linear systems only arise in ARKODE.
Return value	<i>ier</i> is a <i>int</i> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SPGMRSetPrecType for complete further documentation of this routine.

FSUNSPGMRSETMAXRS

Call	FSUNSPGMRSETMAXRS(<i>code</i> , <i>maxrs</i> , <i>ier</i>)
Description	The function FSUNSPGMRSETMAXRS can be called for Fortran programs to change the maximum number of restarts allowed for SPGMR.
Arguments	<i>code</i> (<i>int*</i>) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE). <i>maxrs</i> (<i>int*</i>) maximum allowed number of restarts.
Return value	<i>ier</i> is a <i>int</i> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SPGMRSetMaxRestarts for complete further documentation of this routine.

FSUNMASSSPGMRSETMAXRS

Call	FSUNMASSSPGMRSETMAXRS(<i>maxrs</i> , <i>ier</i>)
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 <i>code</i> is not needed since mass matrix linear systems only arise in ARKODE.
Return value	<i>ier</i> is a <i>int</i> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SPGMRSetMaxRestarts for complete further documentation of this routine.

8.9.2 SUNLINSOL_SPGMR description

The SUNLINSOL_SPGMR module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_SPGMR {
    int maxl;
    int pretype;
    int gstype;
    int max_restarts;
    int numiters;
    realtype resnorm;
    long 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;
};

```

These entries of the *content* field contain the following information:

maxl - number of GMRES basis vectors to use (default is 5),

pretype - flag for type of preconditioning to employ (default is none),

gstype - flag for type of Gram-Schmidt orthogonalization (default is modified Gram-Schmidt),

max_restarts - number of GMRES restarts to allow (default is 0),

numiters - number of iterations from the most-recent solve,

resnorm - final linear residual norm from the most-recent solve,

last_flag - last error return flag from an internal function,

ATimes - function pointer to perform Av product,

ATData - pointer to structure for **ATimes**,

Psetup - function pointer to preconditioner setup routine,

Psolve - function pointer to preconditioner solve routine,

PData - pointer to structure for **Psetup** and **Psolve**,

s1, s2 - vector pointers for supplied scaling matrices (default is NULL),

V - the array of Krylov basis vectors $v_1, \dots, v_{\max l+1}$, stored in $V[0], \dots, V[\max l]$. Each v_i is a vector of type NVECTOR.,

Hes - the $(\max l + 1) \times \max l$ Hessenberg matrix. It is stored row-wise so that the (i,j) th element is given by $Hes[i][j]$.,

givens - a length $2*\max l$ array which represents the Givens rotation matrices that arise in the GMRES

algorithm. These matrices are F_0, F_1, \dots, F_j , where $F_i =$

$$\begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & & c_i & -s_i & \\ & & & s_i & c_i & \\ & & & & & 1 \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix},$$

are represented in the **givens** vector as $givens[0] = c_0$, $givens[1] = s_0$, $givens[2] = c_1$, $givens[3] = s_1, \dots, givens[2j] = c_j$, $givens[2j+1] = s_j$.,

xcor - a vector which holds the scaled, preconditioned correction to the initial guess,

yg - a length (maxl+1) array of `realtype` values used to hold “short” vectors (e.g. y and g),

vtemp - temporary vector storage.

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.

The `SUNLINSOL_SPGMR` module defines implementations of all “iterative” linear solver operations listed in Sections 8.0.1-8.0.3:

- `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`

8.10 The SUNLinearSolver_SPGMR implementation

The SPFGMR (Scaled, Preconditioned, Flexible, Generalized Minimum Residual [35]) implementation of the `SUNLINSOL` module provided with SUNDIALS, `SUNLINSOL_SPGMR`, is an iterative linear solver that is designed to be compatible with any `NVECTOR` implementation (serial, threaded, parallel, and user-supplied) 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, FGMRES is specifically designed to work with a changing preconditioner (e.g. from an iterative method).

8.10.1 SUNLINSOL_SPFGMR usage

The header file to include when using this module is `sunlinsol/sunlinsol.spfgmr.h`. The SUNLINSOL_SPFGMR module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolspfgmr` module library.

The module SUNLINSOL_SPFGMR provides the following user-callable routines:

SUNLinSol_SPFGMR

Call `LS = SUNLinSol_SPFGMR(y, pretype, maxl);`

Description The function `SUNLinSol_SPFGMR` creates and allocates memory for a SPFGMR `SUNLinearSolver`.

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). While it is possible to use a right-preconditioned `SUNLINSOL_SPFGMR` object for these packages, this use mode is not supported and may result in inferior performance.

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`.

Notes

SUNLinSol_SPFGMRSetGSType

Call `retval = SUNLinSol_SPFGMRSetGSType(LS, gstype);`

Description The function `SUNLinSol_SPFGMRSetPrecType` sets the type of Gram-Schmidt orthogonalization 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 values are:

- MODIFIED_GS (1)
- CLASSICAL_GS (2)

Any other integer input will result in a failure, returning error code `SUNLS_ILL_INPUT`.

Return value This routine will return with one of the error codes `SUNLS_ILL_INPUT` (illegal `pretype`), `SUNLS_MEM_NULL` (`S` is `NULL`) or `SUNLS_SUCCESS`.

Notes

`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`.

Notes

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

- `SUNSPFGMR`
 Wrapper function for `SUNLinSol_SPFGMR`
- `SUNSPFGMRSetPrecType`
 Wrapper function for `SUNLinSol_SPFGMRSetPrecType`
- `SUNSPFGMRSetGSType`
 Wrapper function for `SUNLinSol_SPFGMRSetGSType`
- `SUNSPFGMRSetMaxRestarts`
 Wrapper function for `SUNLinSol_SPFGMRSetMaxRestarts`

For solvers that include a Fortran 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 `SUNLINSOL_SPFGMR` object.

Arguments `code` (`int*`) is an integer input specifying the solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, and 4 for `ARKODE`).
`pretype` (`int*`) flag indicating desired preconditioning type
`maxl` (`int*`) flag indicating Krylov subspace size

Return value `ier` is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* the `NVECTOR` object has been initialized.

Allowable values for `pretype` and `maxl` 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 SUNLINSOL_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 Gram-Schmidt orthogonalization algorithm.

Arguments *code* (*int**) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
gstype (*int**) flag indicating the desired orthogonalization algorithm.

Return value *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol_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 orthogonalization algorithm for mass matrix linear systems.

Arguments The arguments are identical to FSUNSPFGMRSETGSTYPE above, except that *code* is not needed since mass matrix linear systems only arise in ARKODE.

Return value *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol_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 indication 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(<i>pretype</i> , <i>ier</i>)
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 <i>code</i> is not needed since mass matrix linear systems only arise in ARKODE.
Return value	<i>ier</i> is a <i>int</i> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SPFGMRSetPrecType for complete further documentation of this routine.

FSUNSPFGMRSETMAXRS

Call	FSUNSPFGMRSETMAXRS(<i>code</i> , <i>maxrs</i> , <i>ier</i>)
Description	The function FSUNSPFGMRSETMAXRS can be called for Fortran programs to change the maximum number of restarts allowed for SPFGMR.
Arguments	<i>code</i> (<i>int*</i>) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE). <i>maxrs</i> (<i>int*</i>) maximum allowed number of restarts.
Return value	<i>ier</i> is a <i>int</i> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SPFGMRSetMaxRestarts for complete further documentation of this routine.

FSUNMASSSPFGMRSETMAXRS

Call	FSUNMASSSPFGMRSETMAXRS(<i>maxrs</i> , <i>ier</i>)
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 <i>code</i> is not needed since mass matrix linear systems only arise in ARKODE.
Return value	<i>ier</i> is a <i>int</i> return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_SPFGMRSetMaxRestarts for complete further documentation of this routine.

8.10.2 SUNLINSOL_SPFGMR description

The SUNLINSOL_SPFGMR module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_SPFGMR {
    int maxl;
    int pretype;
    int gstype;
    int max_restarts;
    int numiters;
    realtype resnorm;
    long 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;
};

```

These entries of the *content* field contain the following information:

maxl - number of FGMRES basis vectors to use (default is 5),

pretype - flag for use of preconditioning (default is none),

gstype - flag for type of Gram-Schmidt orthogonalization (default is modified Gram-Schmidt),

max_restarts - number of FGMRES restarts to allow (default is 0),

numiters - number of iterations from the most-recent solve,

resnorm - final linear residual norm from the most-recent solve,

last_flag - last error return flag from an internal function,

ATimes - function pointer to perform Av product,

ATData - pointer to structure for **ATimes**,

Psetup - function pointer to preconditioner setup routine,

Psolve - function pointer to preconditioner solve routine,

PData - pointer to structure for **Psetup** and **Psolve**,

s1, s2 - vector pointers for supplied scaling matrices (default is NULL),

V - the array of Krylov basis vectors $v_1, \dots, v_{\text{maxl}+1}$, stored in $V[0], \dots, V[\text{maxl}]$. Each v_i is a vector of type NVECTOR.,

Z - the array of preconditioned Krylov basis vectors $z_1, \dots, z_{\text{maxl}+1}$, stored in $Z[0], \dots, Z[\text{maxl}]$. Each z_i is a vector of type NVECTOR.,

Hes - the $(\text{maxl} + 1) \times \text{maxl}$ Hessenberg matrix. It is stored row-wise so that the (i,j)th element is given by $\text{Hes}[i][j]$.,

givens - a length $2*\text{maxl}$ array which represents the Givens rotation matrices that arise in the FGM-

RES algorithm. These matrices are F_0, F_1, \dots, F_j , where $F_i =$

$$\begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & & c_i & -s_i & \\ & & & s_i & c_i & \\ & & & & & 1 \\ & & & & & & \ddots \\ & & & & & & & 1 \end{bmatrix},$$

are represented in the **givens** vector as $\text{givens}[0] = c_0, \text{givens}[1] = s_0, \text{givens}[2] = c_1, \text{givens}[3] = s_1, \dots, \text{givens}[2j] = c_j, \text{givens}[2j+1] = s_j$.,

xcor - a vector which holds the scaled, preconditioned correction to the initial guess,
yg - a length (maxl+1) array of **realtype** values used to hold “short” vectors (e.g. y and g),
vtemp - temporary vector storage.

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.

The **SUNLINSOL_SPFGMR** module defines implementations of all “iterative” linear solver operations listed in Sections 8.0.1-8.0.3:

- **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**

8.11 The SUNLinearSolver_SPBCGS implementation

The SPBCGS (Scaled, Preconditioned, Bi-Conjugate Gradient, Stabilized [37]) implementation of the **SUNLINSOL** module provided with SUNDIALS, **SUNLINSOL_SPBCGS**, is an iterative linear solver that is designed to be compatible with any **NVECTOR** implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (**N_VClone**, **N_VDotProd**, **N_VScale**, **N_VLinearSum**, **N_VProd**, **N_VDiv**, and **N_VDestroy**). Unlike the SPGMR and SPFGMR algorithms, SPBCGS requires a fixed amount of memory that does not increase with the number of allowed iterations.

8.11.1 SUNLINSOL_SPBCGS usage

The header file to include when using this module is `sunlinsol/sunlinsol.spbcgs.h`. The SUNLINSOL_SPBCGS module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolspbcgs` module library.

The module SUNLINSOL_SPBCGS provides the following user-callable routines:

SUNLinSol_SPBCGS

Call `LS = SUNLinSol_SPBCGS(y, pretype, maxl);`

Description The function `SUNLinSol_SPBCGS` creates and allocates memory for a SPBCGS `SUNLinearSolver`.

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 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`.

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_SPBCGS object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.

SUNLinSol_SPBCGSSetPrecType

Call `retval = SUNLinSol_SPBCGSSetPrecType(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`.

Notes

SUNLinSol_SPBCGSsetMaxl

Call `retval = SUNLinSol_SPBCGSsetMaxl(LS, maxl);`

Description The function `SUNLinSol_SPBCGSsetMaxl` updates the number of linear solver iterations to allow.

Arguments `LS` (`SUNLinearSolver`) the SUNLINSOL_SPBCGS 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`.

Notes

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

- `SUNSPBCGS`
Wrapper function for `SUNLinSol_SPBCGS`
- `SUNSPBCGSSetPrecType`
Wrapper function for `SUNLinSol_SPBCGSSetPrecType`
- `SUNSPBCGSsetMaxl`
Wrapper function for `SUNLinSol_SPBCGSsetMaxl`

For solvers that include a Fortran 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 `SUNLINSOL_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 `maxl` are the same as for the C function `SUNLinSol_SPBCGS`.

Additionally, when using `ARKODE` with a non-identity mass matrix, the `SUNLINSOL_SPBCGS` module includes a Fortran-callable function for creating a `SUNLinearSolver` mass matrix solver object.

FSUNMASSSPBCGSINIT

Call `FSUNMASSSPBCGSINIT(pretype, maxl, ier)`

Description The function `FSUNMASSSPBCGSINIT` can be called for Fortran programs to create a `SUNLINSOL_SPBCGS` object for mass matrix linear systems.

Arguments `pretype` (`int*`) flag indicating desired preconditioning type
`maxl` (`int*`) flag indicating number of iterations to allow

Return value `ier` is a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* the `NVECTOR` object has been initialized.
Allowable values for `pretype` and `maxl` are the same as for the C function `SUNLinSol_SPBCGS`.

The `SUNLinSol_SPBCGSSetPrecType` and `SUNLinSol_SPBCGSsetMaxl` routines also support Fortran interfaces for the system and mass matrix solvers.

FSUNSPBCGSSETPRECTYPE

Call	FSUNSPBCGSSETPRECTYPE(<i>code</i> , <i>pretype</i> , <i>ier</i>)
Description	The function FSUNSPBCGSSETPRECTYPE can be called for Fortran programs to change the type of preconditioning to use.
Arguments	<i>code</i> (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE). <i>pretype</i> (int*) flag indication the type of preconditioning to use.
Return value	<i>ier</i> 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(<i>pretype</i> , <i>ier</i>)
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 <i>code</i> is not needed since mass matrix linear systems only arise in ARKODE.
Return value	<i>ier</i> 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(<i>code</i> , <i>maxl</i> , <i>ier</i>)
Description	The function FSUNSPBCGSSETMAXL can be called for Fortran programs to change the maximum number of iterations to allow.
Arguments	<i>code</i> (int*) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE). <i>maxl</i> (int*) the number of iterations to allow
Return value	<i>ier</i> 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(<i>maxl</i> , <i>ier</i>)
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 <i>code</i> is not needed since mass matrix linear systems only arise in ARKODE.
Return value	<i>ier</i> 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.

8.11.2 SUNLINSOL_SPBCGS description

The SUNLINSOL_SPBCGS module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_SPBCGS {
    int maxl;
    int pretype;
    int numiters;
    realtype resnorm;
    long 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;
};
```

These entries of the *content* field contain the following information:

- maxl** - number of SPBCGS iterations to allow (default is 5),
- pretype** - flag for type of preconditioning to employ (default is none),
- numiters** - number of iterations from the most-recent solve,
- resnorm** - final linear residual norm from the most-recent solve,
- last_flag** - last error return flag from an internal function,
- ATimes** - function pointer to perform Av product,
- ATData** - pointer to structure for **ATimes**,
- Psetup** - function pointer to preconditioner setup routine,
- Psolve** - function pointer to preconditioner solve routine,
- PData** - pointer to structure for **Psetup** and **Psolve**,
- s1, s2** - vector pointers for supplied scaling matrices (default is NULL),
- r** - a NVECTOR which holds the current scaled, preconditioned linear system residual,
- r_star** - a NVECTOR which holds the initial scaled, preconditioned linear system residual,
- p, q, u, Ap, vtemp** - NVECTORS used for workspace by the SPBCGS algorithm.

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.

The SUNLINSOL_SPBCGS module defines implementations of all “iterative” linear solver operations listed in Sections 8.0.1-8.0.3:

- `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`

8.12 The SUNLinearSolver_SPTFQMR implementation

The SPTFQMR (Scaled, Preconditioned, Transpose-Free Quasi-Minimum Residual [19]) implementation of the SUNLINSOL module provided with SUNDIALS, `SUNLINSOL_SPTFQMR`, is an iterative linear solver that is designed to be compatible with any `NVECTOR` implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (`N_VClone`, `N_VDotProd`, `N_VScale`, `N_VLinearSum`, `N_VProd`, `N_VConst`, `N_VDiv`, and `N_VDestroy`). Unlike the SPGMR and SPFGMR algorithms, SPTFQMR requires a fixed amount of memory that does not increase with the number of allowed iterations.

8.12.1 SUNLINSOL_SPTFQMR usage

The header file to include when using this module is `sunlinsol/sunlinsol_sptfqmr.h`. The `SUNLINSOL_SPTFQMR` module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolsptfqmr` module library.

The module `SUNLINSOL_SPTFQMR` provides the following user-callable routines:

SUNLinSol_SPTFQMR

Call	<code>LS = SUNLinSol_SPTFQMR(y, pretype, maxl);</code>
Description	The function <code>SUNLinSol_SPTFQMR</code> creates and allocates memory for a SPTFQMR <code>SUNLinearSolver</code> .
Arguments	<p><code>y</code> (<code>N_Vector</code>) a template for cloning vectors needed within the solver</p> <p><code>pretype</code> (<code>int</code>) flag indicating the desired type of preconditioning, allowed values are:</p> <ul style="list-style-type: none"> • <code>PREC_NONE</code> (0) • <code>PREC_LEFT</code> (1) • <code>PREC_RIGHT</code> (2) • <code>PREC_BOTH</code> (3) <p>Any other integer input will result in the default (no preconditioning).</p> <p><code>maxl</code> (<code>int</code>) the number of linear iterations to allow; values ≤ 0 will result in the default value (5).</p>
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>y</code> is incompatible then this routine will return <code>NULL</code> .
Notes	<p>This routine will perform consistency checks to ensure that it is called with a consistent <code>NVECTOR</code> implementation (i.e. that it supplies the requisite vector operations). If <code>y</code> is incompatible, then this routine will return <code>NULL</code>.</p> <p>We note that some SUNDIALS solvers are designed to only work with left preconditioning (<code>IDA</code> and <code>IDAS</code>) and others with only right preconditioning (<code>KINSOL</code>). While it is possible to configure a <code>SUNLINSOL_SPTFQMR</code> object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.</p>

SUNLinSol_SPTFQMRSetPrecType

Call	<code>retval = SUNLinSol_SPTFQMRSetPrecType(LS, pretype);</code>
Description	The function <code>SUNLinSol_SPTFQMRSetPrecType</code> updates the type of preconditioning to use in the <code>SUNLINSOL_SPTFQMR</code> object.
Arguments	<p><code>LS</code> (<code>SUNLinearSolver</code>) the <code>SUNLINSOL_SPTFQMR</code> object to update</p> <p><code>pretype</code> (<code>int</code>) flag indicating the desired type of preconditioning, allowed values match those discussed in <code>SUNLinSol_SPTFQMR</code>.</p>
Return value	This routine will return with one of the error codes <code>SUNLS_ILL_INPUT</code> (illegal <code>pretype</code>), <code>SUNLS_MEM_NULL</code> (<code>S</code> is <code>NULL</code>) or <code>SUNLS_SUCCESS</code> .
Notes	

SUNLinSol_SPTFQMRSetMaxl

Call	<code>retval = SUNLinSol_SPTFQMRSetMaxl(LS, maxl);</code>
Description	The function <code>SUNLinSol_SPTFQMRSetMaxl</code> updates the number of linear solver iterations to allow.
Arguments	<p><code>LS</code> (<code>SUNLinearSolver</code>) the <code>SUNLINSOL_SPTFQMR</code> object to update</p> <p><code>maxl</code> (<code>int</code>) flag indicating the number of iterations to allow; values ≤ 0 will result in the default value (5)</p>
Return value	This routine will return with one of the error codes <code>SUNLS_MEM_NULL</code> (<code>S</code> is <code>NULL</code>) or <code>SUNLS_SUCCESS</code> .
Notes	

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

- **SUNSPTFQMR**
Wrapper function for `SUNLinSol_SPTFQMR`
- **SUNSPTFQMRSetPrecType**
Wrapper function for `SUNLinSol_SPTFQMRSetPrecType`
- **SUNSPTFQMRSetMaxl**
Wrapper function for `SUNLinSol_SPTFQMRSetMaxl`

For solvers that include a Fortran interface module, the `SUNLINSOL_SPTFQMR` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

FSUNSPTFQMRINIT

Call `FSUNSPTFQMRINIT(code, pretype, maxl, ier)`

Description The function `FSUNSPTFQMRINIT` can be called for Fortran programs to create a `SUNLINSOL_SPTFQMR` object.

Arguments `code` (`int*`) is an integer input specifying the solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, and 4 for `ARKODE`).
`pretype` (`int*`) flag indicating desired preconditioning type
`maxl` (`int*`) flag indicating number of iterations to allow

Return value `ier` is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* the `NVECTOR` object has been initialized.

Allowable values for `pretype` and `maxl` are the same as for the C function `SUNLinSol_SPTFQMR`.

Additionally, when using `ARKODE` with a non-identity mass matrix, the `SUNLINSOL_SPTFQMR` module includes a Fortran-callable function for creating a `SUNLinearSolver` mass matrix solver object.

FSUNMASSSPTFQMRINIT

Call `FSUNMASSSPTFQMRINIT(pretype, maxl, ier)`

Description The function `FSUNMASSSPTFQMRINIT` can be called for Fortran programs to create a `SUNLINSOL_SPTFQMR` object for mass matrix linear systems.

Arguments `pretype` (`int*`) flag indicating desired preconditioning type
`maxl` (`int*`) flag indicating number of iterations to allow

Return value `ier` is a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes This routine must be called *after* the `NVECTOR` object has been initialized.

Allowable values for `pretype` and `maxl` are the same as for the C function `SUNLinSol_SPTFQMR`.

The `SUNLinSol_SPTFQMRSetPrecType` and `SUNLinSol_SPTFQMRSetMaxl` routines also support Fortran interfaces for the system and mass matrix solvers.

FSUNSPTFQMRSETPRECTYPE

Call `FSUNSPTFQMRSETPRECTYPE(code, pretype, ier)`

Description The function `FSUNSPTFQMRSETPRECTYPE` can be called for Fortran programs to change the type of preconditioning to use.

Arguments `code` (`int*`) is an integer input specifying the solver id (1 for `CVODE`, 2 for `IDA`, 3 for `KINSOL`, and 4 for `ARKODE`).
`pretype` (`int*`) flag indication the type of preconditioning to use.

Return value `ier` is a `int` return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See `SUNLinSol_SPTFQMRSetPrecType` for complete further documentation of this routine.

FSUNMASSSPTFQMRSETPRECTYPE

Call `FSUNMASSSPTFQMRSETPRECTYPE(prectype, 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 routine.

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.

8.12.2 SUNLINSOL_SPTFQMR description

The `SUNLINSOL_SPTFQMR` module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_SPTFQMR {
    int maxl;
    int pretype;
    int numiters;
    realtype resnorm;
    long 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;
};

```

These entries of the *content* field contain the following information:

maxl - number of TFQMR iterations to allow (default is 5),

pretype - flag for type of preconditioning to employ (default is none),

numiters - number of iterations from the most-recent solve,

resnorm - final linear residual norm from the most-recent solve,

last_flag - last error return flag from an internal function,

ATimes - function pointer to perform Av product,

ATData - pointer to structure for **ATimes**,

Psetup - function pointer to preconditioner setup routine,

Psolve - function pointer to preconditioner solve routine,

PData - pointer to structure for **Psetup** and **Psolve**,

s1, s2 - vector pointers for supplied scaling matrices (default is NULL),

r_star - a NVECTOR which holds the initial scaled, preconditioned linear system residual,

q, d, v, p, u - NVECTORS used for workspace by the SPTFQMR algorithm,

r - array of two NVECTORS used for workspace within the SPTFQMR algorithm,

vtemp1, vtemp2, vtemp3 - temporary vector storage.

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.

The `SUNLINSOL_SPTFQMR` module defines implementations of all “iterative” linear solver operations listed in Sections 8.0.1-8.0.3:

- `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`

8.13 The SUNLinearSolver_PCG implementation

The PCG (Preconditioned Conjugate Gradient [20]) implementation of the `SUNLINSOL` module provided with SUNDIALS, `SUNLINSOL_PCG`, is an iterative linear solver that is designed to be compatible with any `NVECTOR` implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (`N_VClone`, `N_VDotProd`, `N_VScale`, `N_VLinearSum`, `N_VProd`, and `N_VDestroy`). Unlike the SPGMR and SPFGMR algorithms, PCG requires a fixed amount of memory that does not increase with the number of allowed iterations.

Unlike all of the other iterative linear solvers supplied with SUNDIALS, PCG should only be used on *symmetric* linear systems (e.g. mass matrix linear systems encountered in ARKODE). As a result, the explanation of the role of scaling and preconditioning matrices given in general must be modified in this scenario. The PCG algorithm solves a linear system $Ax = b$ where A is a symmetric ($A^T = A$), real-valued matrix. Preconditioning is allowed, and is applied in a symmetric fashion on both the right and left. Scaling is also allowed and is applied symmetrically. We denote the preconditioner and scaling matrices as follows:

- P is the preconditioner (assumed symmetric),
- S is a diagonal matrix of scale factors.

The matrices A and P are not required explicitly; only routines that provide A and P^{-1} as operators are required. The diagonal of the matrix S is held in a single NVECTOR, supplied by the user.

In this notation, PCG applies the underlying CG algorithm to the equivalent transformed system

$$\tilde{A}\tilde{x} = \tilde{b} \quad (8.3)$$

where

$$\begin{aligned} \tilde{A} &= SP^{-1}AP^{-1}S, \\ \tilde{b} &= SP^{-1}b, \\ \tilde{x} &= S^{-1}Px. \end{aligned} \quad (8.4)$$

The scaling matrix must be chosen so that the vectors $SP^{-1}b$ and $S^{-1}Px$ have dimensionless components.

The stopping test for the PCG iterations is on the L2 norm of the scaled preconditioned residual:

$$\begin{aligned} \|\tilde{b} - \tilde{A}\tilde{x}\|_2 &< \delta \\ \Leftrightarrow \\ \|SP^{-1}b - SP^{-1}Ax\|_2 &< \delta \\ \Leftrightarrow \\ \|P^{-1}b - P^{-1}Ax\|_S &< \delta \end{aligned}$$

where $\|v\|_S = \sqrt{v^T S^T S v}$, with an input tolerance δ .

8.13.1 SUNLINSOL_PCG usage

The header file to include when using this module is `sunlinsol/sunlinsol_pcg.h`. The SUNLINSOL_PCG module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolpcg` module library.

The module SUNLINSOL_PCG provides the following user-callable routines:

SUNLinSol_PCG	
Call	<code>LS = SUNLinSol_PCG(y, pretype, maxl);</code>
Description	The function <code>SUNLinSol_PCG</code> creates and allocates memory for a PCG <code>SUNLinearSolver</code> .
Arguments	<p><code>y</code> (N_Vector) a template for cloning vectors needed within the solver</p> <p><code>pretype</code> (int) flag indicating whether to use preconditioning. Since the PCG algorithm is designed to only support symmetric preconditioning, then any of the <code>pretype</code> inputs <code>PREC_LEFT</code> (1), <code>PREC_RIGHT</code> (2), or <code>PREC_BOTH</code> (3) will result in use of the symmetric preconditioner; any other integer input will result in the default (no preconditioning).</p> <p><code>maxl</code> (int) the number of linear iterations to allow; values ≤ 0 will result in the default value (5).</p>
Return value	This returns a <code>SUNLinearSolver</code> object. If either <code>y</code> is incompatible then this routine will return <code>NULL</code> .
Notes	<p>This routine will perform consistency checks to ensure that it is called with a consistent NVECTOR implementation (i.e. that it supplies the requisite vector operations). If <code>y</code> is incompatible, then this routine will return <code>NULL</code>.</p> <p>Although some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL), PCG should <i>only</i> be used with these packages when the linear systems are known to be <i>symmetric</i>. Since the scaling of matrix rows and columns must be identical in a symmetric matrix, symmetric preconditioning should work appropriately even for packages designed with one-sided preconditioning in mind.</p>

SUNLinSol_PCGSetPrecType

Call	<code>retval = SUNLinSol_PCGSetPrecType(LS, pretype);</code>
Description	The function <code>SUNLinSol_PCGSetPrecType</code> updates the flag indicating use of preconditioning in the <code>SUNLINSOL_PCG</code> object.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) the <code>SUNLINSOL_PCG</code> object to update <code>pretype</code> (<code>int</code>) flag indicating use of preconditioning. allowable values match those discussed in <code>SUNLinSol_PCG</code> .
Return value	This routine will return with one of the error codes <code>SUNLS_ILL_INPUT</code> (illegal <code>pretype</code>), <code>SUNLS_MEM_NULL</code> (<code>S</code> is <code>NULL</code>) or <code>SUNLS_SUCCESS</code> .
Notes	

SUNLinSol_PCGSetMaxl

Call	<code>retval = SUNLinSol_PCGSetMaxl(LS, maxl);</code>
Description	The function <code>SUNLinSol_PCGSetMaxl</code> updates the number of linear solver iterations to allow.
Arguments	<code>LS</code> (<code>SUNLinearSolver</code>) the <code>SUNLINSOL_PCG</code> object to update <code>maxl</code> (<code>int</code>) flag indicating the number of iterations to allow; values ≤ 0 will result in the default value (5)
Return value	This routine will return with one of the error codes <code>SUNLS_MEM_NULL</code> (<code>S</code> is <code>NULL</code>) or <code>SUNLS_SUCCESS</code> .
Notes	

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

- `SUNPCG`
 Wrapper function for `SUNLinSol_PCG`
- `SUNPCGSetPrecType`
 Wrapper function for `SUNLinSol_PCGSetPrecType`
- `SUNPCGSetMaxl`
 Wrapper function for `SUNLinSol_PCGSetMaxl`

For solvers that include a Fortran interface module, the `SUNLINSOL_PCG` module also includes a Fortran-callable function for creating a `SUNLinearSolver` object.

FSUNPCGINIT

Call	<code>FSUNPCGINIT(code, pretype, maxl, ier)</code>
Description	The function <code>FSUNPCGINIT</code> can be called for Fortran programs to create a <code>SUNLINSOL_PCG</code> object.
Arguments	<code>code</code> (<code>int*</code>) is an integer input specifying the solver id (1 for <code>CVODE</code> , 2 for <code>IDA</code> , 3 for <code>KINSOL</code> , and 4 for <code>ARKODE</code>). <code>pretype</code> (<code>int*</code>) flag indicating desired preconditioning type <code>maxl</code> (<code>int*</code>) flag indicating number of iterations to allow
Return value	<code>ier</code> is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	This routine must be called <i>after</i> the <code>NVECTOR</code> object has been initialized. Allowable values for <code>pretype</code> and <code>maxl</code> are the same as for the C function <code>SUNLinSol_PCG</code> .

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 indication the type of preconditioning to use.

Return value *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol_PCGSetPrecType for complete further documentation of this routine.

FSUNMASSPCGSETPRECTYPE

Call FSUNMASSPCGSETPRECTYPE(*pretype*, *ier*)

Description The function FSUNMASSPCGSETPRECTYPE can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.

Arguments The arguments are identical to FSUNPCGSETPRECTYPE above, except that *code* is not needed since mass matrix linear systems only arise in ARKODE.

Return value *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol_PCGSetPrecType for complete further documentation of this routine.

FSUNPCGSETMAXL

Call FSUNPCGSETMAXL(*code*, *maxl*, *ier*)

Description The function FSUNPCGSETMAXL can be called for Fortran programs to change the maximum number of iterations to allow.

Arguments *code* (*int**) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).
maxl (*int**) the number of iterations to allow

Return value *ier* is a *int* return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Notes See SUNLinSol_PCGSetMaxl for complete further documentation of this routine.

FSUNMASSPCGSETMAXL

Call	FSUNMASSPCGSETMAXL(maxl, ier)
Description	The function FSUNMASSPCGSETMAXL can be called for Fortran programs to change the type of preconditioning for mass matrix linear systems.
Arguments	The arguments are identical to FSUNPCGSETMAXL above, except that code is not needed since mass matrix linear systems only arise in ARKODE.
Return value	ier is a int return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.
Notes	See SUNLinSol_PCGSetMaxl for complete further documentation of this routine.

8.13.2 SUNLINSOL_PCG description

The SUNLINSOL_PCG module defines the *content* field of a **SUNLinearSolver** to be the following structure:

```
struct _SUNLinearSolverContent_PCG {
    int maxl;
    int pretype;
    int numiters;
    realtype resnorm;
    long 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;
};
```

These entries of the *content* field contain the following information:

maxl - number of PCG iterations to allow (default is 5),
pretype - flag for use of preconditioning (default is none),
numiters - number of iterations from the most-recent solve,
resnorm - final linear residual norm from the most-recent solve,
last_flag - last error return flag from an internal function,
ATimes - function pointer to perform Av product,
ATData - pointer to structure for **ATimes**,
Psetup - function pointer to preconditioner setup routine,
Psolve - function pointer to preconditioner solve routine,
PData - pointer to structure for **Psetup** and **Psolve**,
s - vector pointer for supplied scaling matrix (default is **NULL**),
r - a **NVECTOR** which holds the preconditioned linear system residual,

p, **z**, **Ap** - NVECTORS used for workspace by the PCG algorithm.

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.

The SUNLINSOL_PCG module defines implementations of all “iterative” linear solver operations listed in Sections [8.0.1-8.0.3](#):

- **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**

8.14 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 $10*\text{tol}$, and overwrites x with y prior to returning (in case the calling routine would like to investigate further).
- **Test_SUNLinSolSetATimes** (iterative solvers only): Verifies that **SUNLinSolSetATimes** can be called and returns successfully.
- **Test_SUNLinSolSetPreconditioner** (iterative solvers only): Verifies that **SUNLinSolSetPreconditioner** can be called and returns successfully.
- **Test_SUNLinSolSetScalingVectors** (iterative solvers only): Verifies that **SUNLinSolSetScalingVectors** can be called and returns successfully.
- **Test_SUNLinSolLastFlag**: Verifies that **SUNLinSolLastFlag** can be called, and outputs the result to **stdout**.
- **Test_SUNLinSolNumIters** (iterative solvers only): Verifies that **SUNLinSolNumIters** can be called, and outputs the result to **stdout**.
- **Test_SUNLinSolResNorm** (iterative solvers only): Verifies that **SUNLinSolResNorm** can be called, and that the result is non-negative.
- **Test_SUNLinSolResid** (iterative solvers only): Verifies that **SUNLinSolResid** can be called.
- **Test_SUNLinSolSpace** verifies that **SUNLinSolSpace** can be called, and outputs the results to **stdout**.

We'll note that these tests should be performed in a particular order. For either direct or iterative linear solvers, **Test_SUNLinSolInitialize** must be called before **Test_SUNLinSolSetup**, which must be called before **Test_SUNLinSolSolve**. Additionally, for iterative linear solvers **Test_SUNLinSolSetATimes**, **Test_SUNLinSolSetPreconditioner** and **Test_SUNLinSolSetScalingVectors** should be called before **Test_SUNLinSolInitialize**; similarly **Test_SUNLinSolNumIters**, **Test_SUNLinSolResNorm** and **Test_SUNLinSolResid** should be called after **Test_SUNLinSolSolve**. These are called in the appropriate order in all of the example problems.

8.15 SUNLinearSolver functions used by CVODE

In Table 8.3, we list the linear solver functions in the **SUNLINSOL** module used within the **CVLS** interface in the **CVODE** package. In general, **CVLS** considers two non-overlapping categories of linear solvers: *matrix-based* and *matrix-free*, determined based on whether the **SUNMATRIX** object **J** passed to **CVodeSetLinearSolver** was not **NULL**.

Additionally, **CVLS** will consider a linear solver of either type as *iterative* if it self-identifies as **SUNLINEARSOLVER_ITERATIVE** (via the **SUNLinSolGetType** routine). Since both matrix-based and matrix-free linear solvers may be iterative, we only list **SUNLINSOL** routines that are specifically called based on this type; these routines are *in addition to* those listed for the other two categories.

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 ✓ to indicate that they are required, or with † to indicate that they are only called if they are non-**NULL** in the **SUNLINSOL** implementation that is being used.

Table 8.3: List of linear solver functions usage by CVODE code modules

	Matrix-Based	Matrix-Free	Iterative
SUNLinSolGetType	✓	✓	
SUNLinSolSetATimes		✓	
SUNLinSolSetPreconditioner			†
SUNLinSolSetScalingVectors	†	†	
SUNLinSolInitialize	✓	✓	
SUNLinSolSetup	✓	✓	
SUNLinSolSolve	✓	✓	
¹ SUNLinSolNumIters			†
² SUNLinSolLastFlag			
SUNLinSolFree	✓	✓	
SUNLinSolSpace	†	†	

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.

Chapter 9

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.

The time integrators in SUNDIALS specify a default nonlinear solver module and as such this chapter is intended for users that wish to use a non-default nonlinear solver module or would like to provide their own nonlinear solver implementation. Users interested in using a non-default solver module may skip the description of the SUNNONLINSOL API in section 9.1 and proceed 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 9.1.1 – 9.1.3. This is followed by the definition of functions supplied to a nonlinear solver implementation in section 9.1.4. A table of nonlinear solver return codes is given in section 9.1.5. The `SUNNonlinearSolver` type and the generic SUNNONLINSOL module are defined in section 9.1.6. Section 9.1.7 describes how SUNNONLINSOL models interface with SUNDIALS integrators providing sensitivity analysis capabilities (CVODES and IDAS). Finally, section 9.1.8 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.

9.1 The SUNNonlinearSolver API

The SUNNONLINSOL API defines several nonlinear solver operations that enable SUNDIALS integrators to utilize any SUNNONLINSOL implementation that provides the required functions. These functions can be divided into three categories. The first are the core nonlinear solver functions. The second 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`.

9.1.1 SUNNonlinearSolver core functions

The core nonlinear solver functions consist of two required functions to get the nonlinear solver type (`SUNNonlinSolGetType`) and solve the nonlinear system (`SUNNonlinSolSolve`). 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 $G(y) = y$.

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 initialization may set this operation to `NULL`.

SUNNonlinSolSetup

Call `retval = SUNNonlinSolSetup(NLS, y, mem);`

Description The *optional* function `SUNNonlinSolSetup` performs any solver setup needed for a nonlinear 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 `SUNNonlinSolSetup` before each step attempt. `SUNNONLINSOL` implementations that do not require setup may set this operation to `NULL`.

SUNNonlinSolSolve

Call `retval = SUNNonlinSolSolve(NLS, y0, y, 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 initial iterate for the nonlinear solve. This *must* remain unchanged throughout the solution process.

`y` (`N_Vector`) the solution to the nonlinear system.

`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 successful solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.

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`.

9.1.2 SUNNonlinearSolver 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 9.1.4 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.

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 9.1.4 for the definition of `SUNNonlinLSetupFn`.

Return value The return value `retval` (of type `int`) should be zero for a successful call, and a negative value for a failure.

Notes The `SUNNonlinLSetupFn` function sets up the linear system $Ax = b$ where $A = \frac{\partial F}{\partial y}$ is the linearization of the nonlinear residual function $F(y) = 0$ (when using `SUNLINSOL` direct linear solvers) or calls the user-defined preconditioner setup function (when using `SUNLINSOL` iterative linear solvers). `SUNNONLINSOL` implementations that do not require solving this system, do not utilize `SUNLINSOL` linear solvers, or use `SUNLINSOL` linear solvers that do not require setup may set this operation to `NULL`.

SUNNonlinSolSetLSolveFn

Call	<code>retval = SUNNonlinSolSetLSolveFn(NLS, LSolveFn);</code>
Description	The <i>optional</i> function <code>SUNNonlinSolSetLSolveFn</code> is called by SUNDIALS integrators to provide the nonlinear solver with access to its linear solver solve function.
Arguments	<code>NLS</code> (<code>SUNNonlinearSolver</code>) a <code>SUNNONLINSOL</code> object <code>LSolveFn</code> (<code>SUNNonlinSolLSolveFn</code>) a wrapper function to the SUNDIALS integrator's linear solver solve function. See section 9.1.4 for the definition of <code>SUNNonlinSolLSolveFn</code> .
Return value	The return value <code>retval</code> (of type <code>int</code>) should be zero for a successful call, and a negative value for a failure.
Notes	The <code>SUNNonlinLSolveFn</code> function solves the linear system $Ax = b$ where $A = \frac{\partial F}{\partial y}$ is the linearization of the nonlinear residual function $F(y) = 0$. <code>SUNNONLINSOL</code> implementations that do not require solving this system or do not use <code>SUNLINSOL</code> linear solvers may set this operation to <code>NULL</code> .

SUNNonlinSolSetConvTestFn

Call	<code>retval = SUNNonlinSolSetConvTestFn(NLS, CTestFn);</code>
Description	The <i>optional</i> function <code>SUNNonlinSolSetConvTestFn</code> is used to provide the nonlinear solver with a function for determining if the nonlinear solver iteration has converged. This is typically called by SUNDIALS integrators to define their nonlinear convergence criteria, but may be replaced by the user.
Arguments	<code>NLS</code> (<code>SUNNonlinearSolver</code>) a <code>SUNNONLINSOL</code> object. <code>CTestFn</code> (<code>SUNNonlineSolConvTestFn</code>) a SUNDIALS integrator's nonlinear solver convergence test function. See section 9.1.4 for the definition of <code>SUNNonlinSolConvTestFn</code> .
Return value	The return value <code>retval</code> (of type <code>int</code>) should be zero for a successful call, and a negative value for a failure.
Notes	<code>SUNNONLINSOL</code> implementations utilizing their own convergence test criteria may set this function to <code>NULL</code> .

SUNNonlinSolSetMaxIters

Call	<code>retval = SUNNonlinSolSetMaxIters(NLS, maxiters);</code>
Description	The <i>optional</i> function <code>SUNNonlinSolSetMaxIters</code> sets the maximum number of nonlinear solver iterations. This is typically called by SUNDIALS integrators to define their default iteration limit, but may be adjusted by the user.
Arguments	<code>NLS</code> (<code>SUNNonlinearSolver</code>) a <code>SUNNONLINSOL</code> object. <code>maxiters</code> (<code>int</code>) the maximum number of nonlinear iterations.
Return value	The return value <code>retval</code> (of type <code>int</code>) should be zero for a successful call, and a negative value for a failure (e.g., <code>maxiters < 1</code>).

9.1.3 SUNNonlinearSolver get functions

The following get functions allow SUNDIALS integrators to retrieve nonlinear solver statistics. The routine for getting the current total number of iterations (`SUNNonlinSolGetNumIters` is optional. The routine for getting 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	<code>retval = SUNNonlinSolGetNumIters(NLS, numiters);</code>
Description	The <i>optional</i> function <code>SUNNonlinSolGetNumIters</code> returns the total number of nonlinear solver iterations. This is typically called by the SUNDIALS integrator to store the nonlinear solver statistics, but may also be called by the user.
Arguments	<code>NLS</code> (<code>SUNNonlinearSolver</code>) a <code>SUNNONLINSOL</code> object <code>numiters</code> (<code>long int*</code>) the total number of nonlinear solver iterations.
Return value	The return value <code>retval</code> (of type <code>int</code>) should be zero for a successful call, and a negative value for a failure.

SUNNonlinSolGetCurIter

Call	<code>retval = SUNNonlinSolGetCurIter(NLS, iter);</code>
Description	The function <code>SUNNonlinSolGetCurIter</code> returns the iteration index of the current nonlinear solve. This function is <i>required</i> when using SUNDIALS integrator-provided convergence tests or when using a <code>SUNLINSOL</code> spils linear solver; otherwise it is <i>optional</i> .
Arguments	<code>NLS</code> (<code>SUNNonlinearSolver</code>) a <code>SUNNONLINSOL</code> object <code>iter</code> (<code>int*</code>) the nonlinear solver iteration in the current solve starting from zero.
Return value	The return value <code>retval</code> (of type <code>int</code>) should be zero for a successful call, and a negative value for a failure.

9.1.4 Functions provided by SUNDIALS integrators

To interface with `SUNNONLINSOL` modules, the SUNDIALS integrators supply a variety of routines for evaluating the nonlinear system, calling the `SUNLINSOL` setup and solve functions, and testing the nonlinear iteration for convergence. These integrator-provided routines translate between the user-supplied ODE or DAE systems and the generic interfaces to the nonlinear or linear systems of equations that result in their solution. The types for functions provided to a `SUNNONLINSOL` module are defined in the header file `sundials/sundials_nonlinearsolver.h`, and are described below.

SUNNonlinSolSysFn

Definition	<code>typedef int (*SUNNonlinSolSysFn)(N_Vector y, N_Vector F, void* mem);</code>
Purpose	These functions evaluate the nonlinear system $F(y)$ for <code>SUNNONLINEARSOLVER_ROOTFIND</code> type modules or $G(y)$ for <code>SUNNONLINEARSOLVER_FIXEDPOINT</code> type modules. Memory for <code>F</code> must be allocated prior to calling this function. The vector <code>y</code> <i>must</i> be left unchanged.
Arguments	<code>y</code> is the state vector at which the nonlinear system should be evaluated. <code>F</code> is the output vector containing $F(y)$ or $G(y)$, depending on the solver type. <code>mem</code> is the SUNDIALS integrator memory structure.
Return value	The return value <code>retval</code> (of type <code>int</code>) is zero for a successful solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.

SUNNonlinSolLSetupFn

Definition	<code>typedef int (*SUNNonlinSolLSetupFn)(N_Vector y, N_Vector F, booleantype jbad, booleantype* jcur, void* mem);</code>
Purpose	These functions are wrappers to the SUNDIALS integrator's function for setting up linear solves with <code>SUNLINSOL</code> modules.

Arguments	<p>y is the state vector at which the linear system should be setup.</p> <p>F is the value of the nonlinear system function at y.</p> <p>jbad is an input indicating whether the nonlinear solver believes that A has gone stale (SUNTRUE) or not (SUNFALSE).</p> <p>jcur is an output indicating whether the routine has updated the Jacobian A (SUNTRUE) or not (SUNFALSE).</p> <p>mem is the SUNDIALS integrator memory structure.</p>
Return value	The return value retval (of type int) is zero for a successful 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.

SUNNonlinSolLSolveFn

Definition	<code>typedef int (*SUNNonlinSolLSolveFn)(N_Vector y, N_Vector b, void* mem);</code>
Purpose	These functions are wrappers to the SUNDIALS integrator's function for solving linear systems with SUNLINSOL modules.
Arguments	<p>y is the input vector containing the current nonlinear iteration.</p> <p>b contains the right-hand side vector for the linear solve on input and the solution to the linear system on output.</p> <p>mem is the SUNDIALS integrator memory structure.</p>
Return value	The return value retval (of type int) is zero for a successful 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.

SUNNonlinSolConvTestFn

Definition	<code>typedef int (*SUNNonlinSolConvTestFn)(SUNNonlinearSolver NLS, N_Vector y, N_Vector del, realtype tol, N_Vector ewt, void* mem);</code>
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	<p>NLS is the SUNNONLINSOL object.</p> <p>y is the current nonlinear iterate.</p> <p>del is the difference between the current and prior nonlinear iterates.</p> <p>tol is the nonlinear solver tolerance.</p> <p>ewt is the weight vector used in computing weighted norms.</p> <p>mem is the SUNDIALS integrator memory structure.</p>
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 <code>ewt</code> . SUNNONLINSOL modules utilizing their own convergence criteria may ignore these functions.	

9.1.5 SUNNonlinearSolver return codes

The functions provided to SUNNONLINSOL modules by each SUNDIALS integrator, and functions within the SUNDIALS-provided SUNNONLINSOL implementations utilize a common set of return codes, shown below in Table 9.1. Here, negative values correspond to non-recoverable failures, positive values to recoverable failures, and zero to a successful call.

Table 9.1: Description of the SUNNonlinearSolver return codes

Name	Value	Description
SUN-NLS_SUCCESS	0	successful call or converged solve
SUN-NLS_CONTINUE	1	the nonlinear solver is not converged, keep iterating
SUN-NLS_CONV_RECVR	2	the nonlinear solver appears to be diverging, try to recover
SUN-NLS_MEM_NULL	-1	a memory argument is NULL
SUN-NLS_MEM_FAIL	-2	a memory access or allocation failed
SUN-NLS_ILL_INPUT	-3	an illegal input option was provided

9.1.6 The generic SUNNonlinearSolver module

SUNDIALS integrators interact with specific SUNNONLINSOL implementations through the generic SUNNONLINSOL module on which all other SUNNONLINSOL implementations are built. The `SUNNonlinearSolver` type is a pointer to a structure containing an implementation-dependent *content* field and an *ops* field. The type `SUNNonlinearSolver` is defined as follows:

```
typedef struct _generic_SUNNonlinearSolver *SUNNonlinearSolver;
```

```
struct _generic_SUNNonlinearSolver {
    void *content;
    struct _generic_SUNNonlinearSolver_Ops *ops;
};
```

where the `_generic_SUNNonlinearSolver_Ops` structure is a list of pointers to the various actual nonlinear solver operations provided by a specific implementation. The `_generic_SUNNonlinearSolver_Ops` structure is defined as

```
struct _generic_SUNNonlinearSolver_Ops {
    SUNNonlinearSolver_Type (*gettype)(SUNNonlinearSolver);
    int (*initialize)(SUNNonlinearSolver);
    int (*setup)(SUNNonlinearSolver, N_Vector, void*);
    int (*solve)(SUNNonlinearSolver, N_Vector, N_Vector,
                 N_Vector, realtype, booleantype, void*);
    int (*free)(SUNNonlinearSolver);
    int (*setsysfn)(SUNNonlinearSolver, SUNNonlinSolSysFn);
    int (*setlssetupfn)(SUNNonlinearSolver, SUNNonlinSolLSSetupFn);
    int (*setlsolvefn)(SUNNonlinearSolver, SUNNonlinSolLSolveFn);
    int (*setctestfn)(SUNNonlinearSolver, SUNNonlinSolConvTestFn);
    int (*setmaxiters)(SUNNonlinearSolver, int);
```

```

    int                (*getnumiters)(SUNNonlinearSolver, long int*);
    int                (*getcuriter)(SUNNonlinearSolver, int*);
};

```

The generic SUNNONLINSOL module defines and implements the nonlinear solver operations defined in Sections 9.1.1 – 9.1.3. These routines are in fact only wrappers to the nonlinear solver operations provided by a particular SUNNONLINSOL implementation, which are accessed through the ops field of the SUNNonlinearSolver structure. To illustrate this point we show below the implementation of a typical nonlinear solver operation from the generic SUNNONLINSOL module, namely SUNNonlinSolSolve, which solves the nonlinear system and returns a flag denoting a successful or failed solve:

```

int SUNNonlinSolSolve(SUNNonlinearSolver NLS,
                     N_Vector y0, N_Vector y,
                     N_Vector w, realtype tol,
                     booleantype callSetup, void* mem)
{
    return((int) NLS->ops->solve(NLS, y0, y, w, tol, callSetup, mem));
}

```

9.1.7 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 6.2 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

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

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_VDotProd(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_VWrmsNorm(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	<code>w = N_VNewEmpty_SensWrapper(count);</code>
Description	The function <code>N_VNewEmpty_SensWrapper</code> creates an empty NVECTOR_SENSWRAPPER wrapper with space for <code>count</code> vectors.
Arguments	<code>count</code> (<code>int</code>) the number of vectors the wrapper will contain.
Return value	The return value <code>w</code> (of type <code>N_Vector</code>) will be a NVECTOR object if the constructor exits successfully, otherwise <code>w</code> will be <code>NULL</code> .

`N_VNew_SensWrapper`

Call	<code>w = N_VNew_SensWrapper(count, y);</code>
Description	The function <code>N_VNew_SensWrapper</code> creates an NVECTOR_SENSWRAPPER wrapper containing <code>count</code> vectors cloned from <code>y</code> .
Arguments	<code>count</code> (<code>int</code>) the number of vectors the wrapper will contain. <code>y</code> (<code>N_Vector</code>) the template vectors to use in creating the vector wrapper.
Return value	The return value <code>w</code> (of type <code>N_Vector</code>) will be a NVECTOR object if the constructor exits successfully, otherwise <code>w</code> will be <code>NULL</code> .

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;
    boolean_t 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

9.1.8 Implementing a Custom SUNNonlinearSolver 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 9.1.1 – 9.1.3. Note that the names of the module routines should be unique to that implementation in order to permit using more than one SUNNONLINSOL module (each with different `SUNNonlinearSolver` internal data representations) in the same code.

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.

9.2 The `SUNNonlinearSolver_Newton` implementation

This section describes the `SUNNONLINSOL` implementation of Newton’s method. To access the `SUNNONLINSOL_NEWTON` module, include the header file `sunnonlinsol/sunnonlinsol_newton.h`. We note that the `SUNNONLINSOL_NEWTON` module is accessible from `SUNDIALS` integrators *without* separately linking to the `libsundials_sunnonlinsolnewton` module library.

9.2.1 `SUNNonlinearSolver_Newton` description

To find the solution to

$$F(y) = 0 \tag{9.1}$$

given an initial guess $y^{(0)}$, Newton’s method computes a series of approximate solutions

$$y^{(m+1)} = y^{(m)} + \delta^{(m+1)} \tag{9.2}$$

where m is the Newton iteration index, and the Newton update $\delta^{(m+1)}$ is the solution of the linear system

$$A(y^{(m)})\delta^{(m+1)} = -F(y^{(m)}), \tag{9.3}$$

in which A is the Jacobian matrix

$$A \equiv \partial F / \partial y. \tag{9.4}$$

Depending on the linear solver used, the `SUNNONLINSOL_NEWTON` module will employ either a Modified Newton method, or an Inexact Newton method [4, 7, 15, 17, 30]. 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 `SUNNonlinSolSetupFn` 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 `SUNNonlinSolSetupFn` function in two instances:

- (a) when requested by the integrator (the input `callSetSetup` 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 `SUNNonlinSolSetupFn` 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.

9.2.2 SUNNonlinearSolver_Newton functions

The `SUNNONLINSOL_NEWTON` module provides the following constructors for creating a `SUNNonlinearSolver` object.

`SUNNonlinSol_Newton`

Call	<code>NLS = SUNNonlinSol_Newton(y);</code>
Description	The function <code>SUNNonlinSol_Newton</code> creates a <code>SUNNonlinearSolver</code> object for use with SUNDIALS integrators to solve nonlinear systems of the form $F(y) = 0$ using Newton's method.
Arguments	<code>y</code> (<code>N_Vector</code>) a template for cloning vectors needed within the solver.
Return value	The return value <code>NLS</code> (of type <code>SUNNonlinearSolver</code>) will be a <code>SUNNONLINSOL</code> object if the constructor exits successfully, otherwise <code>NLS</code> will be <code>NULL</code> .

`SUNNonlinSol_NewtonSens`

Call	<code>NLS = SUNNonlinSol_NewtonSens(count, y);</code>
Description	The function <code>SUNNonlinSol_NewtonSens</code> creates a <code>SUNNonlinearSolver</code> 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	<p><code>count</code> (<code>int</code>) the number of vectors in the nonlinear solve. When integrating a system containing <code>Ns</code> sensitivities the value of <code>count</code> is:</p> <ul style="list-style-type: none"> • <code>Ns+1</code> if using a <i>simultaneous</i> corrector approach. • <code>Ns</code> if using a <i>staggered</i> corrector approach. <p><code>y</code> (<code>N_Vector</code>) a template for cloning vectors needed within the solver.</p>
Return value	The return value <code>NLS</code> (of type <code>SUNNonlinearSolver</code>) will be a <code>SUNNONLINSOL</code> object if the constructor exits successfully, otherwise <code>NLS</code> will be <code>NULL</code> .

The `SUNNONLINSOL_NEWTON` module implements all of the functions defined in sections 9.1.1 – 9.1.3 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 9.1.1 – 9.1.3 should be called in favor of the `SUNNONLINSOL_NEWTON`-specific implementations.

The `SUNNONLINSOL_NEWTON` module also defines the following additional user-callable function.

`SUNNonlinSolGetSysFn_Newton`

Call	<code>retval = SUNNonlinSolGetSysFn_Newton(NLS, SysFn);</code>
Description	The function <code>SUNNonlinSolGetSysFn_Newton</code> returns the residual function that defines the nonlinear system.
Arguments	<p><code>NLS</code> (<code>SUNNonlinearSolver</code>) a <code>SUNNONLINSOL</code> object</p> <p><code>SysFn</code> (<code>SUNNonlinSolSysFn*</code>) the function defining the nonlinear system.</p>
Return value	The return value <code>retval</code> (of type <code>int</code>) 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 <code>SUNNONLINSOL_NEWTON</code> module. We note that <code>SUNNONLINSOL_NEWTON</code> will not leverage the results from any user calls to <code>SysFn</code> .

9.2.3 SUNNonlinearSolver_Newton content

The *content* field of the SUNNONLINSOL_NEWTON module is the following structure.

```
struct _SUNNonlinearSolverContent_Newton {

    SUNNonlinSolSysFn    Sys;
    SUNNonlinSolLSetupFn LSetup;
    SUNNonlinSolLSolveFn LSolve;
    SUNNonlinSolConvTestFn CTest;

    N_Vector    delta;
    booleantype jcur;
    int         curiter;
    int         maxiters;
    long int    niters;
};
```

These entries of the *content* field contain the following information:

Sys - the function for evaluating the nonlinear system,
LSetup - the package-supplied function for setting up the linear solver,
LSolve - the package-supplied function for performing a linear solve,
CTest - the function for checking convergence of the Newton iteration,
delta - the Newton iteration update vector,
jcur - the Jacobian status (SUNTRUE = current, SUNFALSE = stale),
curiter - the current number of iterations in the solve attempt,
maxiters - the maximum number of Newton iterations allowed in a solve, and
niters - the total number of nonlinear iterations across all solves.

9.2.4 SUNNonlinearSolver_Newton Fortran interface

For SUNDIALS integrators that include a Fortran interface, the SUNNONLINSOL_NEWTON module also includes a Fortran-callable function for creating a **SUNNonlinearSolver** object.

FSUNNEWTONINIT

Call **FSUNNEWTONINIT**(code, ier);

Description The function **FSUNNEWTONINIT** can be called for Fortran programs to create a **SUNNonlinearSolver** object for use with SUNDIALS integrators to solve nonlinear systems 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, 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.

9.3 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.

9.3.1 SUNNonlinearSolver_FixedPoint description

To find the solution to

$$G(y) = y \quad (9.5)$$

given an initial guess $y^{(0)}$, the fixed point iteration computes a series of approximate solutions

$$y^{(n+1)} = G(y^{(n)}) \quad (9.6)$$

where n is the iteration index. The convergence of this iteration may be accelerated using Anderson's method [3, 38, 18, 32]. With Anderson acceleration using subspace size m , the series of approximate solutions can be formulated as the linear combination

$$y^{(n+1)} = \sum_{i=0}^{m_n} \alpha_i^{(n)} G(y^{(n-m_n+i)}) \quad (9.7)$$

where $m_n = \min\{m, n\}$ and the factors

$$\alpha^{(n)} = (\alpha_0^{(n)}, \dots, \alpha_{m_n}^{(n)}) \quad (9.8)$$

solve the minimization problem $\min_{\alpha} \|F_n \alpha^T\|_2$ under the constraint that $\sum_{i=0}^{m_n} \alpha_i = 1$ where

$$F_n = (f_{n-m_n}, \dots, f_n) \quad (9.9)$$

with $f_i = G(y^{(i)}) - y^{(i)}$. Due to this constraint, in the limit of $m = 0$ the accelerated fixed point iteration formula (9.7) simplifies to the standard fixed point iteration (9.6).

Following the recommendations made in [38], the SUNNONLINSOL_FIXEDPOINT implementation computes the series of approximate solutions as

$$y^{(n+1)} = G(y^{(n)}) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta g_{n-m_n+i} \quad (9.10)$$

with $\Delta g_i = G(y^{(i+1)}) - G(y^{(i)})$ and where the factors

$$\gamma^{(n)} = (\gamma_0^{(n)}, \dots, \gamma_{m_n-1}^{(n)}) \quad (9.11)$$

solve the unconstrained minimization problem $\min_{\gamma} \|f_n - \Delta F_n \gamma^T\|_2$ where

$$\Delta F_n = (\Delta f_{n-m_n}, \dots, \Delta f_{n-1}), \quad (9.12)$$

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.

9.3.2 SUNNonlinearSolver_FixedPoint functions

The SUNNONLINSOL_FIXEDPOINT module provides the following constructors for creating a `SUNNonlinearSolver` object.

SUNNonlinSol_FixedPoint

Call	<code>NLS = SUNNonlinSol_FixedPoint(y, m);</code>
Description	The function <code>SUNNonlinSol_FixedPoint</code> creates a <code>SUNNonlinearSolver</code> object for use with SUNDIALS integrators to solve nonlinear systems of the form $G(y) = y$.
Arguments	<code>y</code> (<code>N_Vector</code>) a template for cloning vectors needed within the solver <code>m</code> (<code>int</code>) the number of acceleration vectors to use
Return value	The return value <code>NLS</code> (of type <code>SUNNonlinearSolver</code>) will be a <code>SUNNONLINSOL</code> object if the constructor exits successfully, otherwise <code>NLS</code> will be <code>NULL</code> .

SUNNonlinSol_FixedPointSens

Call	<code>NLS = SUNNonlinSol_FixedPointSens(count, y, m);</code>
Description	The function <code>SUNNonlinSol_FixedPointSens</code> creates a <code>SUNNonlinearSolver</code> object for use with SUNDIALS sensitivity enabled integrators (CVODES and IDAS) to solve nonlinear systems of the form $G(y) = y$.
Arguments	<code>count</code> (<code>int</code>) the number of vectors in the nonlinear solve. When integrating a system containing <code>Ns</code> sensitivities the value of <code>count</code> is: <ul style="list-style-type: none"> • <code>Ns+1</code> if using a <i>simultaneous</i> corrector approach. • <code>Ns</code> if using a <i>staggered</i> corrector approach. <code>y</code> (<code>N_Vector</code>) a template for cloning vectors needed within the solver. <code>m</code> (<code>int</code>) the number of acceleration vectors to use.
Return value	The return value <code>NLS</code> (of type <code>SUNNonlinearSolver</code>) will be a <code>SUNNONLINSOL</code> object if the constructor exits successfully, otherwise <code>NLS</code> will be <code>NULL</code> .

Since the accelerated fixed point iteration (9.6) does not require the setup or solution of any linear systems, the `SUNNONLINSOL_FIXEDPOINT` module implements all of the functions defined in sections 9.1.1 – 9.1.3 except for the `SUNNonlinSolSetup`, `SUNNonlinSolSetLSetupFn`, and `SUNNonlinSolSetLSolveFn` functions, that are set to `NULL`. The `SUNNONLINSOL_FIXEDPOINT` functions have the same names as those defined by the generic `SUNNONLINSOL` API with `_FixedPoint` appended to the function name. Unless using the `SUNNONLINSOL_FIXEDPOINT` module as a standalone nonlinear solver the generic functions defined in sections 9.1.1 – 9.1.3 should be called in favor of the `SUNNONLINSOL_FIXEDPOINT`-specific implementations.

The `SUNNONLINSOL_FIXEDPOINT` module also defines the following additional user-callable function.

SUNNonlinSolGetSysFn_FixedPoint

Call	<code>retval = SUNNonlinSolGetSysFn_FixedPoint(NLS, SysFn);</code>
Description	The function <code>SUNNonlinSolGetSysFn_FixedPoint</code> returns the fixed-point function that defines the nonlinear system.
Arguments	<code>NLS</code> (<code>SUNNonlinearSolver</code>) a <code>SUNNONLINSOL</code> object <code>SysFn</code> (<code>SUNNonlinSolSysFn*</code>) the function defining the nonlinear system.
Return value	The return value <code>retval</code> (of type <code>int</code>) 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 <code>SUNNONLINSOL_FIXEDPOINT</code> module. We note that <code>SUNNONLINSOL_FIXEDPOINT</code> will not leverage the results from any user calls to <code>SysFn</code> .

9.3.3 SUNNonlinearSolver_FixedPoint content

The *content* field of the SUNNONLINSOL_FIXEDPOINT module is the following structure.

```
struct _SUNNonlinearSolverContent_FixedPoint {

    SUNNonlinSolSysFn    Sys;
    SUNNonlinSolConvTestFn CTest;

    int      m;
    int      *imap;
    realtype *R;
    realtype *gamma;
    realtype *cvals;
    N_Vector *df;
    N_Vector *dg;
    N_Vector *q;
    N_Vector *Xvecs;
    N_Vector yprev;
    N_Vector gy;
    N_Vector fold;
    N_Vector gold;
    N_Vector delta;
    int      curiter;
    int      maxiters;
    long int niters;
};
```

The following entries of the *content* field are always allocated:

Sys - function for evaluating the nonlinear system,
 CTest - function for checking convergence of the fixed point iteration,
 yprev - N_Vector used to store previous fixed-point iterate,
 gy - N_Vector used to store $G(y)$ in fixed-point algorithm,
 delta - N_Vector used to store difference between successive fixed-point iterates,
 curiter - the current number of iterations in the solve attempt,
 maxiters - the maximum number of fixed-point iterations allowed in a solve, and
 niters - the total number of nonlinear iterations across all solves.

m - number of acceleration vectors,

If Anderson acceleration is requested (i.e., $m > 0$ in the call to SUNNonlinSol_FixedPoint), then the following items are also allocated within the *content* field:

imap - index array used in acceleration algorithm (length m)
 R - small matrix used in acceleration algorithm (length m*m)
 gamma - small vector used in acceleration algorithm (length m)
 cvals - small vector used in acceleration algorithm (length m+1)
 df - array of N_Vectors used in acceleration algorithm (length m)
 dg - array of N_Vectors used in acceleration algorithm (length m)
 q - array of N_Vectors used in acceleration algorithm (length m)
 Xvecs - N_Vector pointer array used in acceleration algorithm (length m+1)
 fold - N_Vector used in acceleration algorithm
 gold - N_Vector used in acceleration algorithm

9.3.4 SUNNonlinearSolver_FixedPoint Fortran interface

For SUNDIALS integrators that include a Fortran interface, the SUNNONLINSOL_FIXEDPOINT module also includes a Fortran-callable function for creating a `SUNNonlinearSolver` object.

FSUNFIXEDPOINTINIT

Call	<code>FSUNFIXEDPOINTINIT(code, m, ier);</code>
Description	The function <code>FSUNFIXEDPOINTINIT</code> can be called for Fortran programs to create a <code>SUNNonlinearSolver</code> object for use with SUNDIALS integrators to solve nonlinear systems of the form $G(y) = y$.
Arguments	<p><code>code</code> (<code>int*</code>) is an integer input specifying the solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, and 4 for ARKODE).</p> <p><code>m</code> (<code>int*</code>) is an integer input specifying the number of acceleration vectors.</p>
Return value	<code>ier</code> is a return completion flag equal to 0 for a success return and -1 otherwise. See printed message for details in case of failure.

Appendix A

SUNDIALS Package Installation Procedure

The installation of any SUNDIALS package is accomplished by installing the SUNDIALS suite as a whole, according to the instructions that follow. The same procedure applies whether or not the downloaded file contains one or all solvers in SUNDIALS.

The SUNDIALS suite (or individual solvers) are distributed as compressed archives (`.tar.gz`). The name of the distribution archive is of the form `solver-x.y.z.tar.gz`, where *solver* is one of: `sundials`, `cvode`, `cvodes`, `arkode`, `ida`, `idas`, or `kinsol`, and `x.y.z` represents the version number (of the SUNDIALS suite or of the individual solver). To begin the installation, first uncompress and expand the sources, by issuing

```
% tar xzf solver-x.y.z.tar.gz
```

This will extract source files under a directory `solver-x.y.z`.

Starting with version 2.6.0 of SUNDIALS, CMake is the only supported method of installation. The explanations of the installation procedure begins with a few common observations:

- The remainder of this chapter will follow these conventions:

solverdir is the directory `solver-x.y.z` created above; i.e., the directory containing the SUNDIALS sources.

builddir is the (temporary) directory under which SUNDIALS is built.

instdir is the directory under which the SUNDIALS exported header files and libraries will be installed. Typically, header files are exported under a directory `instdir/include` while libraries are installed under `instdir/lib`, with *instdir* specified at configuration time.

- For SUNDIALS CMake-based installation, in-source builds are prohibited; in other words, the build directory *builddir* can **not** be the same as *solverdir* and such an attempt will lead to an error. This prevents “polluting” the source tree and allows efficient builds for different configurations and/or options.
- The installation directory *instdir* can **not** be the same as the source directory *solverdir*.
- By default, only the libraries and header files are exported to the installation directory *instdir*. If enabled by the user (with the appropriate toggle for CMake), the examples distributed with SUNDIALS will be built together with the solver libraries but the installation step will result in exporting (by default in a subdirectory of the installation directory) the example sources and sample outputs together with automatically generated configuration files that reference the *installed* SUNDIALS headers and libraries. As such, these configuration files for the SUNDIALS examples can be used as “templates” for your own problems. CMake installs `CMakeLists.txt` files and also (as an option available only under Unix/Linux) `Makefile` files. Note this installation



approach also allows the option of building the SUNDIALS examples without having to install them. (This can be used as a sanity check for the freshly built libraries.)

- Even if generation of shared libraries is enabled, only static libraries are created for the FCMIX modules. (Because of the use of fixed names for the Fortran user-provided subroutines, FCMIX shared libraries would result in “undefined symbol” errors at link time.)

A.1 CMake-based installation

CMake-based installation provides a platform-independent build system. CMake can generate Unix and Linux Makefiles, as well as KDevelop, Visual Studio, and (Apple) XCode project files from the same configuration file. In addition, CMake also provides a GUI front end and which allows an interactive build and installation process.

The SUNDIALS build process requires CMake version 3.1.3 or higher and a working C compiler. On Unix-like operating systems, it also requires Make (and **curses**, including its development libraries, for the GUI front end to CMake, **ccmake**), while on Windows it requires Visual Studio. CMake is continually adding new features, and the latest version can be downloaded from <http://www.cmake.org>. Build instructions for CMake (only necessary for Unix-like systems) can be found on the CMake website. Once CMake is installed, Linux/Unix users will be able to use **ccmake**, while Windows users will be able to use **CMakeSetup**.

As previously noted, when using CMake to configure, build and install SUNDIALS, it is always required to use a separate build directory. While in-source builds are possible, they are explicitly prohibited by the SUNDIALS CMake scripts (one of the reasons being that, unlike autotools, CMake does not provide a **make distclean** procedure and it is therefore difficult to clean-up the source tree after an in-source build). By ensuring a separate build directory, it is an easy task for the user to clean-up all traces of the build by simply removing the build directory. CMake does generate a **make clean** which will remove files generated by the compiler and linker.

A.1.1 Configuring, building, and installing on Unix-like systems

The default CMake configuration will build all included solvers and associated examples and will build static and shared libraries. The *instdir* defaults to */usr/local* and can be changed by setting the **CMAKE_INSTALL_PREFIX** variable. Support for FORTRAN and all other options are disabled.

CMake can be used from the command line with the **cmake** command, or from a **curses**-based GUI by using the **ccmake** command. Examples for using both methods will be presented. For the examples shown it is assumed that there is a top level SUNDIALS directory with appropriate source, build and install directories:

```
% mkdir (...)sundials/instdir
% mkdir (...)sundials/builddir
% cd (...)sundials/builddir
```

Building with the GUI

Using CMake with the GUI follows this general process:

- Select and modify values, run configure (c key)
- New values are denoted with an asterisk
- To set a variable, move the cursor to the variable and press enter
 - If it is a boolean (ON/OFF) it will toggle the value
 - If it is string or file, it will allow editing of the string
 - For file and directories, the <tab> key can be used to complete

- Repeat until all values are set as desired and the generate option is available (g key)
- Some variables (advanced variables) are not visible right away
- To see advanced variables, toggle to advanced mode (t key)
- To search for a variable press / key, and to repeat the search, press the n key

To build the default configuration using the GUI, from the *builddir* enter the `ccmake` command and point to the *solverdir*:

```
% ccmake ../solverdir
```

The default configuration screen is shown in Figure A.1.

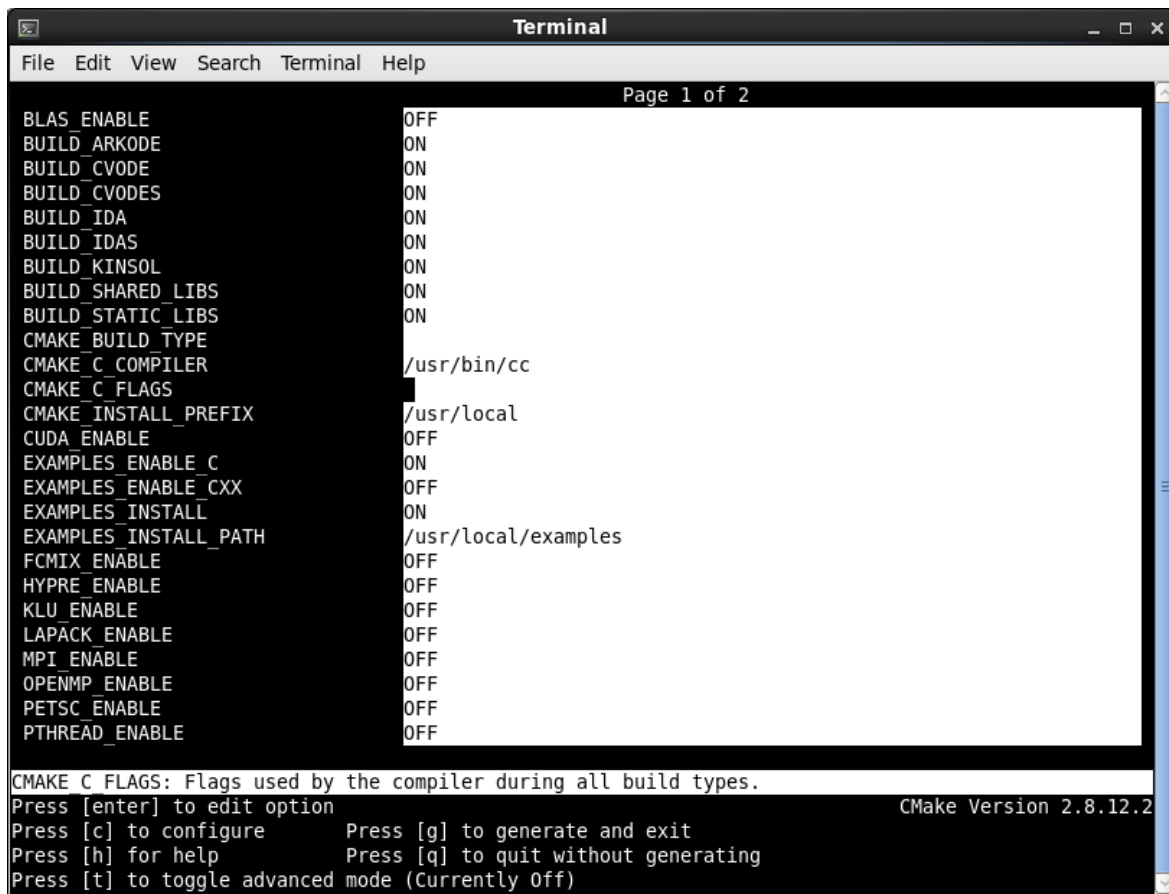


Figure A.1: Default configuration screen. Note: Initial screen is empty. To get this default configuration, press 'c' repeatedly (accepting default values denoted with asterisk) until the 'g' option is available.

The default *instldir* 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
```

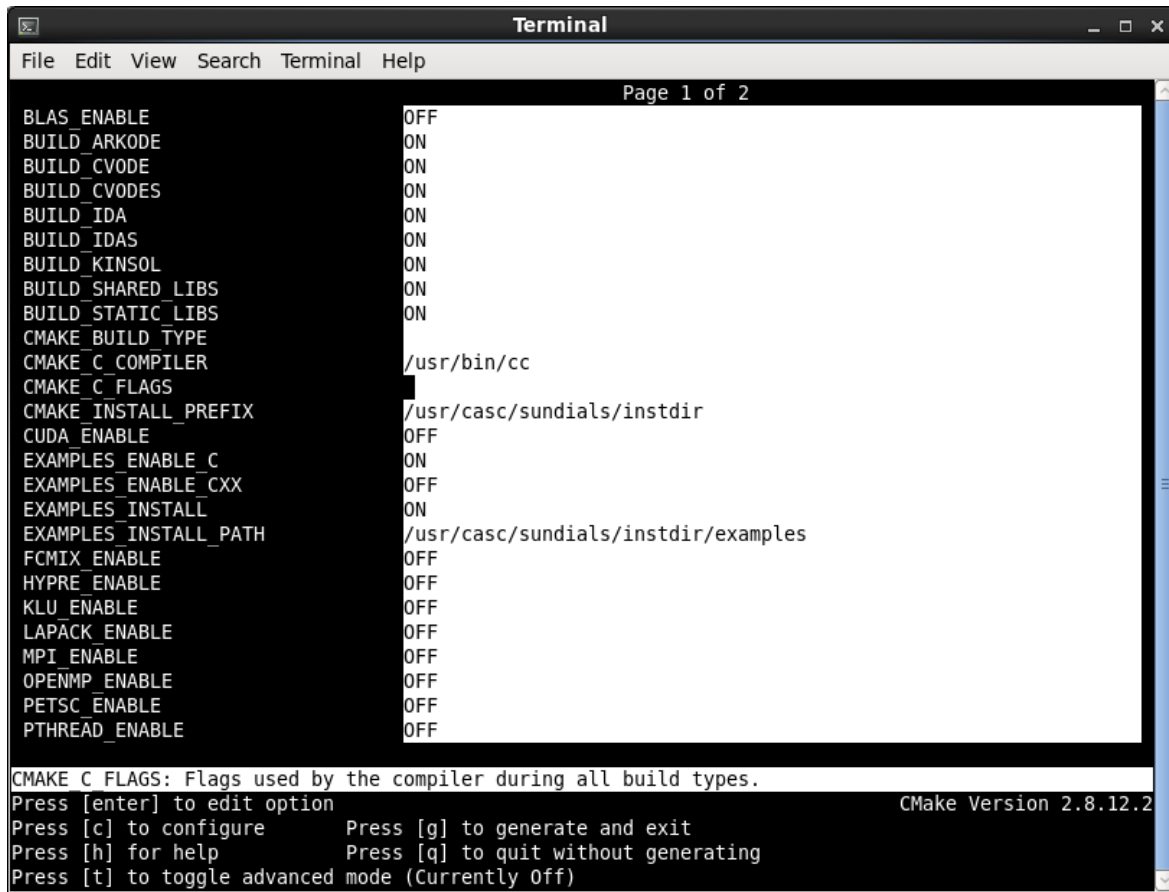


Figure A.2: Changing the *instdir* for SUNDIALS and corresponding examples

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.

BLAS_ENABLE - Enable BLAS support

Default: OFF

Note: Setting this option to ON will trigger additional CMake options. See additional information on building with BLAS enabled in [A.1.4](#).

BLAS_LIBRARIES - BLAS library

Default: /usr/lib/libblas.so

Note: CMake will search for libraries in your `LD_LIBRARY_PATH` prior to searching default system paths.

`BUILD_ARKODE` - Build the ARKODE library
Default: ON

`BUILD_CVODE` - Build the CVODE library
Default: ON

`BUILD_CVODES` - Build the CVODES library
Default: ON

`BUILD_IDA` - Build the IDA library
Default: ON

`BUILD_IDAS` - Build the IDAS library
Default: ON

`BUILD_KINSOL` - Build the KINSOL library
Default: ON

`BUILD_SHARED_LIBS` - Build shared libraries
Default: ON

`BUILD_STATIC_LIBS` - Build static libraries
Default: ON

`CMAKE_BUILD_TYPE` - Choose the type of build, options are: `None` (`CMAKE_C_FLAGS` used), `Debug`, `Release`, `RelWithDebInfo`, and `MinSizeRel`
Default:
Note: Specifying a build type will trigger the corresponding build type specific compiler flag options below which will be appended to the flags set by `CMAKE_<language>_FLAGS`.

`CMAKE_C_COMPILER` - C compiler
Default: `/usr/bin/cc`

`CMAKE_C_FLAGS` - Flags for C compiler
Default:

`CMAKE_C_FLAGS_DEBUG` - Flags used by the C compiler during debug builds
Default: `-g`

`CMAKE_C_FLAGS_MINSIZEREL` - Flags used by the C compiler during release minsize builds
Default: `-Os -DNDEBUG`

`CMAKE_C_FLAGS_RELEASE` - Flags used by the C compiler during release builds
Default: `-O3 -DNDEBUG`

`CMAKE_CXX_COMPILER` - C++ compiler
Default: `/usr/bin/c++`
Note: A C++ compiler (and all related options) are only triggered if C++ examples are enabled (`EXAMPLES_ENABLE_CXX` is ON). All SUNDIALS solvers can be used from C++ applications by default without setting any additional configuration options.

`CMAKE_CXX_FLAGS` - Flags for C++ compiler
Default:

`CMAKE_CXX_FLAGS_DEBUG` - Flags used by the C++ compiler during debug builds
Default: `-g`

CMAKE_CXX_FLAGS_MINSIZEREL - Flags used by the C++ compiler during release minsize builds

Default: -Os -DNDEBUG

CMAKE_CXX_FLAGS_RELEASE - Flags used by the C++ compiler during release builds

Default: -O3 -DNDEBUG

CMAKE_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 BLAS/LAPACK support is enabled (**BLAS_ENABLE** or **LAPACK_ENABLE** is ON).

CMAKE_Fortran_FLAGS - Flags for Fortran compiler

Default:

CMAKE_Fortran_FLAGS_DEBUG - Flags used by the Fortran compiler during debug builds

Default: -g

CMAKE_Fortran_FLAGS_MINSIZEREL - Flags used by the Fortran compiler during release minsize builds

Default: -Os

CMAKE_Fortran_FLAGS_RELEASE - Flags used by the Fortran compiler during release builds

Default: -O3

CMAKE_INSTALL_PREFIX - Install path prefix, prepended onto install directories

Default: /usr/local

Note: The user must have write access to the location specified through this option. Exported SUNDIALS header files and libraries will be installed under subdirectories **include** and **lib** of **CMAKE_INSTALL_PREFIX**, respectively.

CUDA_ENABLE - Build the SUNDIALS CUDA vector module.

Default: OFF

EXAMPLES_ENABLE_C - Build the SUNDIALS C examples

Default: ON

EXAMPLES_ENABLE_CUDA - Build the SUNDIALS CUDA examples

Default: OFF

Note: You need to enable CUDA support to build these examples.

EXAMPLES_ENABLE_CXX - Build the SUNDIALS C++ examples

Default: OFF

EXAMPLES_ENABLE_RAJA - Build the SUNDIALS RAJA examples

Default: OFF

Note: You need to enable CUDA and RAJA support to build these examples.

EXAMPLES_ENABLE_F77 - Build the SUNDIALS Fortran77 examples

Default: ON (if **FCMIX_ENABLE** is ON)

EXAMPLES_ENABLE_F90 - Build the SUNDIALS Fortran90 examples

Default: OFF

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`.

`FCMIX_ENABLE` - Enable Fortran-C support

Default: `OFF`

`HYPRE_ENABLE` - 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

`KLU_ENABLE` - 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

`LAPACK_ENABLE` - 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.

`MPI_ENABLE` - Enable MPI support (build the parallel nvector).

Default: `OFF`

Note: Setting this option to `ON` will trigger several additional options related to MPI.

`MPI_C_COMPILER` - `mpicc` program

Default:

`MPI_CXX_COMPILER` - `mpicxx` program

Default:

Note: This option is triggered only if MPI is enabled (`MPI_ENABLE` 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 `MPI_ENABLE`.

`MPI_Fortran_COMPILER` - `mpif77` or `mpif90` program

Default:

Note: This option is triggered only if MPI is enabled (`MPI_ENABLE` is `ON`), Fortran-C support is enabled (`FCMIX_ENABLE` is `ON`), and Fortran77 or Fortran90 examples are enabled (`EXAMPLES_ENABLE_F77` or `EXAMPLES_ENABLE_F90` are `ON`).

MPIEXEC_EXECUTABLE - Specify the executable for running MPI programs

Default: `mpirun`

Note: This option is triggered only if MPI is enabled (`MPI_ENABLE` is ON).

OPENMP_ENABLE - Enable OpenMP support (build the OpenMP nvector).

Default: OFF

PETSC_ENABLE - Enable PETSc support

Default: OFF

Note: See additional information on building with PETSc enabled in [A.1.4](#).

PETSC_INCLUDE_DIR - Path to PETSc header files

PETSC_LIBRARY_DIR - Path to PETSc installed library files

PTHREAD_ENABLE - Enable Pthreads support (build the Pthreads nvector).

Default: OFF

RAJA_ENABLE - Enable RAJA support (build the RAJA nvector).

Default: OFF

Note: You need to enable CUDA in order to build the RAJA vector module.

SUNDIALS_F77_FUNC_CASE - **advanced option** - Specify the case to use in the Fortran name-mangling scheme, options are: `lower` or `upper`

Default:

Note: The build system will attempt to infer the Fortran name-mangling scheme using the Fortran compiler. This option should only be used if a Fortran compiler is not available or to override the inferred or default (`lower`) scheme if one can not be determined. If used, `SUNDIALS_F77_FUNC_UNDERSCORES` must also be set.

SUNDIALS_F77_FUNC_UNDERSCORES - **advanced option** - Specify the number of underscores to append in the Fortran name-mangling scheme, options are: `none`, `one`, or `two`

Default:

Note: The build system will attempt to infer the Fortran name-mangling scheme using the Fortran compiler. This option should only be used if a Fortran compiler is not available or to override the inferred or default (`one`) scheme if one can not be determined. If used, `SUNDIALS_F77_FUNC_CASE` must also be set.

SUNDIALS_INDEX_TYPE - **advanced**

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`

SUPERLUMT_ENABLE - Enable SuperLU_MT support

Default: OFF

Note: See additional information on building with SuperLU_MT 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_THREAD_TYPE - Must be set to Pthread or OpenMP

Default: Pthread

USE_GENERIC_MATH - Use generic (stdc) math libraries

Default: ON

xSDK Configuration Options

SUNDIALS supports CMake configuration options defined by the Extreme-scale Scientific Software Development Kit (xSDK) community policies (see <https://xsdk.info> for more information). xSDK CMake options are unused by default but may be activated by setting **USE_XSDK_DEFAULTS** to ON.

When xSDK options are active, they will overwrite the corresponding SUNDIALS option and may have different default values (see details below). As such the equivalent SUNDIALS options should not be used when configuring with xSDK options. In the GUI front end to CMake (**ccmake**), setting **USE_XSDK_DEFAULTS** to ON will hide the corresponding SUNDIALS options as advanced CMake variables. During configuration, messages are output detailing which xSDK flags are active and the equivalent SUNDIALS options that are replaced. Below is a complete list xSDK options and the corresponding SUNDIALS options if applicable.



TPL_BLAS_LIBRARIES - BLAS library

Default: /usr/lib/libblas.so

SUNDIALS equivalent: **BLAS_LIBRARIES**

Note: CMake will search for libraries in your **LD_LIBRARY_PATH** prior to searching default system paths.

TPL_ENABLE_BLAS - Enable BLAS support

Default: OFF

SUNDIALS equivalent: **BLAS_ENABLE**

TPL_ENABLE_HYPRE - Enable *hypre* support

Default: OFF

SUNDIALS equivalent: **HYPRE_ENABLE**

TPL_ENABLE_KLU - Enable KLU support

Default: OFF

SUNDIALS equivalent: **KLU_ENABLE**

TPL_ENABLE_PETSC - Enable PETSc support

Default: OFF

SUNDIALS equivalent: **PETSC_ENABLE**

TPL_ENABLE_LAPACK - Enable LAPACK support

Default: OFF

SUNDIALS equivalent: **LAPACK_ENABLE**

TPL_ENABLE_SUPERLUMT - Enable SuperLU_MT support

Default: OFF

SUNDIALS equivalent: **SUPERLUMT_ENABLE**

TPL_HYPRE_INCLUDE_DIRS - Path to *hypre* header files

SUNDIALS equivalent: **HYPRE_INCLUDE_DIR**

TPL_HYPRE_LIBRARIES - *hypre* library
SUNDIALS equivalent: N/A

TPL_KLU_INCLUDE_DIRS - Path to KLU header files
SUNDIALS equivalent: KLU_INCLUDE_DIR

TPL_KLU_LIBRARIES - KLU library
SUNDIALS equivalent: N/A

TPL_LAPACK_LIBRARIES - LAPACK (and BLAS) libraries
Default: /usr/lib/liblapack.so;/usr/lib/libblas.so
SUNDIALS equivalent: LAPACK_LIBRARIES
Note: CMake will search for libraries in your LD_LIBRARY_PATH prior to searching default system paths.

TPL_PETSC_INCLUDE_DIRS - Path to PETSc header files
SUNDIALS equivalent: PETSC_INCLUDE_DIR

TPL_PETSC_LIBRARIES - PETSc library
SUNDIALS equivalent: N/A

TPL_SUPERLUMT_INCLUDE_DIRS - Path to SuperLU_MT header files
SUNDIALS equivalent: SUPERLUMT_INCLUDE_DIR

TPL_SUPERLUMT_LIBRARIES - SuperLU_MT library
SUNDIALS equivalent: N/A

TPL_SUPERLUMT_THREAD_TYPE - SuperLU_MT library thread type
SUNDIALS equivalent: SUPERLUMT_THREAD_TYPE

USE_XSDK_DEFAULTS - Enable xSDK default configuration settings
Default: OFF
SUNDIALS equivalent: N/A
Note: Enabling xSDK defaults also sets CMAKE_BUILD_TYPE to Debug

XSDK_ENABLE_FORTRAN - Enable SUNDIALS Fortran interface
Default: OFF
SUNDIALS equivalent: FCMIX_ENABLE

XSDK_INDEX_SIZE - Integer size (bits) used for indices in SUNDIALS, options are: 32 or 64
Default: 32
SUNDIALS equivalent: SUNDIALS_INDEX_SIZE

XSDK_PRECISION - Precision used in SUNDIALS, options are: **double**, **single**, or **quad**
Default: **double**
SUNDIALS equivalent: SUNDIALS_PRECISION

A.1.3 Configuration examples

The following examples will help demonstrate usage of the CMake configure options. To configure SUNDIALS using the default C and Fortran compilers, and default **mpicc** and **mpif77** parallel compilers, enable compilation of examples, and install libraries, headers, and example sources under subdirectories of **/home/myname/sundials/**, use:

```
% cmake \  
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \  
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \  
> -DMPI_ENABLE=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 \
> -DMPI_ENABLE=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 external libraries any used with SUNDIALS must also be build as a shared library or as a static library compiled with the `-fPIC` flag.



Building with BLAS

SUNDIALS does not utilize BLAS directly but it may be needed by other external libraries that SUNDIALS can be built with (e.g. LAPACK, PETSc, SuperLU_MT, etc.). To enable BLAS, set the `BLAS_ENABLE` option to `ON`. If the directory containing the BLAS library is in the `LD_LIBRARY_PATH` environment variable, CMake will set the `BLAS_LIBRARIES` variable accordingly, otherwise CMake will attempt to find the BLAS library in standard system locations. To explicitly tell CMake what libraries to use, the `BLAS_LIBRARIES` variable can be set to the desired library. Example:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DBLAS_ENABLE=ON \
> -DBLAS_LIBRARIES=/myblaspath/lib/libblas.so \
> -DSUPERLUMT_ENABLE=ON \
> -DSUPERLUMT_INCLUDE_DIR=/mysuperlumtpath/SRC
> -DSUPERLUMT_LIBRARY_DIR=/mysuperlumtpath/lib
> /home/myname/sundials/solverdir
%
% make install
%
```

When allowing CMake to automatically locate the LAPACK library, CMake *may* also locate the corresponding BLAS library.



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 LAPACK

To enable LAPACK, set the `LAPACK_ENABLE` 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. When setting the LAPACK location explicitly the location of the corresponding BLAS library will also need to be set. Example:



```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DBLAS_ENABLE=ON \
> -DBLAS_LIBRARIES=/mylapackpath/lib/libblas.so \
> -DLAPACK_ENABLE=ON \
> -DLAPACK_LIBRARIES=/mylapackpath/lib/liblapack.so \
> /home/myname/sundials/solverdir
%
% make install
%
```



When allowing CMake to automatically locate the LAPACK library, CMake *may* also locate the corresponding BLAS library.

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 4.5.3. To enable KLU, set `KLU_ENABLE` to `ON`, set `KLU_INCLUDE_DIR` to the `include` path of the KLU installation and set `KLU_LIBRARY_DIR` to the `lib` path of the KLU installation. The CMake configure will result in populating the following variables: `AMD_LIBRARY`, `AMD_LIBRARY_DIR`, `BTF_LIBRARY`, `BTF_LIBRARY_DIR`, `COLAMD_LIBRARY`, `COLAMD_LIBRARY_DIR`, and `KLU_LIBRARY`.

Building with SuperLU_MT

The SuperLU_MT libraries are available for download from the Lawrence Berkeley National Laboratory website: http://crd-legacy.lbl.gov/~xiaoye/SuperLU/#superlu_mt. SUNDIALS has been tested with SuperLU_MT version 3.1. To enable SuperLU_MT, set `SUPERLUMT_ENABLE` 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_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 `OPENMP_ENABLE` or `PTHREAD_ENABLE` set to `ON` then SuperLU_MT should be set to use the same threading type.



Building with PETSc

The PETSc libraries are available for download from the Argonne National Laboratory website: <http://www.mcs.anl.gov/petsc>. SUNDIALS has been tested with PETSc version 3.7.2. To enable PETSc, set `PETSC_ENABLE` to `ON`, set `PETSC_INCLUDE_DIR` to the `include` path of the PETSc installation, and set the variable `PETSC_LIBRARY_DIR` to the `lib` path of the PETSc installation.

Building with *hypre*

The *hypre* libraries are available for download from the Lawrence Livermore National Laboratory website: <http://computation.llnl.gov/projects/hypre>. SUNDIALS has been tested with *hypre* version 2.11.1. To enable *hypre*, set `HYPRE_ENABLE` 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.

Building with CUDA

SUNDIALS CUDA modules and examples have been tested with version 8.0 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 `CUDA_ENABLE` 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 version 0.3. Building SUNDIALS RAJA modules requires a CUDA-enabled RAJA installation. To enable RAJA, set `CUDA_ENABLE` and `RAJA_ENABLE` 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_RAJA` to `ON`.

A.1.5 Testing the build and installation

If SUNDIALS was configured with `EXAMPLES_ENABLE_<language>` options to `ON`, then a set of regression tests can be run after building with the `make` command by running:

```
% make test
```

Additionally, if `EXAMPLES_INSTALL` was also set to `ON`, then a set of smoke tests can be run after installing with the `make install` command by running:

```
% make test_install
```

A.2 Building and Running Examples

Each of the SUNDIALS solvers is distributed with a set of examples demonstrating basic usage. To build and install the examples, set at least of the `EXAMPLES_ENABLE_<language>` options to `ON`, and set `EXAMPLES_INSTALL` to `ON`. Specify the installation path for the examples with the variable `EXAMPLES_INSTALL_PATH`. CMake will generate `CMakeLists.txt` configuration files (and `Makefile` files if on Linux/Unix) that reference the *installed* SUNDIALS headers and libraries.

Either the `CMakeLists.txt` file or the traditional `Makefile` may be used to build the examples as well as serve as a template for creating user developed solutions. To use the supplied `Makefile` simply run `make` to compile and generate the executables. To use CMake from within the installed example directory, run `cmake` (or `ccmake` to use the GUI) followed by `make` to compile the example code. Note that if CMake is used, it will overwrite the traditional `Makefile` with a new CMake-generated `Makefile`. The resulting output from running the examples can be compared with example output bundled in the SUNDIALS distribution.

NOTE: There will potentially be differences in the output due to machine architecture, compiler versions, use of third party libraries etc.



A.3 Configuring, building, and installing on Windows

CMake can also be used to build SUNDIALS on Windows. To build SUNDIALS for use with Visual Studio the following steps should be performed:

1. Unzip the downloaded tar file(s) into a directory. This will be the *solverdir*
2. Create a separate *builddir*
3. Open a Visual Studio Command Prompt and cd to *builddir*
4. Run `cmake-gui ../solverdir`
 - (a) Hit Configure
 - (b) Check/Uncheck solvers to be built
 - (c) Change CMAKE_INSTALL_PREFIX to *instdir*
 - (d) Set other options as desired
 - (e) Hit Generate
5. Back in the VS Command Window:
 - (a) Run `msbuild ALL_BUILD.vcxproj`
 - (b) Run `msbuild INSTALL.vcxproj`

The resulting libraries will be in the *instdir*. The SUNDIALS project can also now be opened in Visual Studio. Double click on the ALL_BUILD.vcxproj file to open the project. Build the whole *solution* to create the SUNDIALS libraries. To use the SUNDIALS libraries in your own projects, you must set the include directories for your project, add the SUNDIALS libraries to your project solution, and set the SUNDIALS libraries as dependencies for your project.

A.4 Installed libraries and exported header files

Using the CMake SUNDIALS build system, the command

```
% make install
```

will install the libraries under *libdir* and the public header files under *includedir*. The values for these directories are *instdir/lib* and *instdir/include*, respectively. The location can be changed by setting the CMake variable CMAKE_INSTALL_PREFIX. Although all installed libraries reside under *libdir/lib*, the public header files are further organized into subdirectories under *includedir/include*.

The installed libraries and exported header files are listed for reference in Table A.1. The file extension *.lib* is typically *.so* for shared libraries and *.a* for static libraries. Note that, in the Tables, names are relative to *libdir* for libraries and to *includedir* for header files.

A typical user program need not explicitly include any of the shared SUNDIALS header files from under the *includedir/include/sundials* directory since they are explicitly included by the appropriate solver header files (*e.g.*, *cvode_dense.h* includes *sundials_dense.h*). However, it is both legal and safe to do so, and would be useful, for example, if the functions declared in *sundials_dense.h* are to be used in building a preconditioner.

Table A.1: SUNDIALS libraries and header files

SHARED	Libraries	n/a
<i>continued on next page</i>		

<i>continued from last page</i>		
	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
NVECTOR_SERIAL	Libraries	libsundials_nvecserial. <i>lib</i> libsundials_fnvecserial.a
	Header files	nvector/nvector_serial.h
NVECTOR_PARALLEL	Libraries	libsundials_nvecparallel. <i>lib</i> libsundials_fnvecparallel.a
	Header files	nvector/nvector_parallel.h
NVECTOR_OPENMP	Libraries	libsundials_nvecopenmp. <i>lib</i> libsundials_fnvecopenmp.a
	Header files	nvector/nvector_openmp.h
NVECTOR_PTHREADS	Libraries	libsundials_nvecpthreads. <i>lib</i> libsundials_fnvecpthreads.a
	Header files	nvector/nvector_pthreads.h
NVECTOR_PARHYP	Libraries	libsundials_nvecparhyp. <i>lib</i>
	Header files	nvector/nvector_parhyp.h
NVECTOR_PETSC	Libraries	libsundials_nvecpetsc. <i>lib</i>
	Header files	nvector/nvector_petsc.h
NVECTOR_CUDA	Libraries	libsundials_nveccuda. <i>lib</i>
	Libraries	libsundials_nvecmpicuda. <i>lib</i>
	Header files	nvector/nvector_cuda.h nvector/nvector_mpicuda.h nvector/cuda/ThreadPartitioning.hpp nvector/cuda/Vector.hpp nvector/cuda/VectorKernels.cuh
NVECTOR_RAJA	Libraries	libsundials_nveccudaraja. <i>lib</i>
	Libraries	libsundials_nveccudampiraja. <i>lib</i>
	Header files	nvector/nvector_raja.h nvector/nvector_mpiraja.h nvector/raja/Vector.hpp
SUNMATRIX_BAND	Libraries	libsundials_sunmatrixband. <i>lib</i> libsundials_fsunmatrixband.a
<i>continued on next page</i>		

<i>continued from last page</i>		
	Header files	sunmatrix/sunmatrix_band.h
SUNMATRIX_DENSE	Libraries	libsundials_sunmatrixdense. <i>lib</i> libsundials_fsunmatrixdense.a
	Header files	sunmatrix/sunmatrix_dense.h
SUNMATRIX_SPARSE	Libraries	libsundials_sunmatrixsparse. <i>lib</i> libsundials_fsunmatrixsparse.a
	Header files	sunmatrix/sunmatrix_sparse.h
SUNLINSOL_BAND	Libraries	libsundials_sunlinsolband. <i>lib</i> libsundials_fsunlinsolband.a
	Header files	sunlinsol/sunlinsol_band.h
SUNLINSOL_DENSE	Libraries	libsundials_sunlinsoldense. <i>lib</i> libsundials_fsunlinsoldense.a
	Header files	sunlinsol/sunlinsol_dense.h
SUNLINSOL_KLU	Libraries	libsundials_sunlinsolklu. <i>lib</i> libsundials_fsunlinsolklu.a
	Header files	sunlinsol/sunlinsol_klu.h
SUNLINSOL_LAPACKBAND	Libraries	libsundials_sunlinsollapackband. <i>lib</i> libsundials_fsunlinsollapackband.a
	Header files	sunlinsol/sunlinsol_lapackband.h
SUNLINSOL_LAPACKDENSE	Libraries	libsundials_sunlinsollapackdense. <i>lib</i> libsundials_fsunlinsollapackdense.a
	Header files	sunlinsol/sunlinsol_lapackdense.h
SUNLINSOL_PCG	Libraries	libsundials_sunlinsolpcg. <i>lib</i> libsundials_fsunlinsolpcg.a
	Header files	sunlinsol/sunlinsol_pcg.h
SUNLINSOL_SPCGGS	Libraries	libsundials_sunlinsolspbcgs. <i>lib</i> libsundials_fsunlinsolspbcgs.a
	Header files	sunlinsol/sunlinsol_spbcgs.h
SUNLINSOL_SPGMR	Libraries	libsundials_sunlinsolspfgmr. <i>lib</i> libsundials_fsunlinsolspfgmr.a
	Header files	sunlinsol/sunlinsol_spfgmr.h
SUNLINSOL_SPGMR	Libraries	libsundials_sunlinsolspgmr. <i>lib</i> libsundials_fsunlinsolspgmr.a
	Header files	sunlinsol/sunlinsol_spgmr.h
SUNLINSOL_SPTFQMR	Libraries	libsundials_sunlinsolsptfqmr. <i>lib</i> libsundials_fsunlinsolsptfqmr.a
	Header files	sunlinsol/sunlinsol_sptfqmr.h
SUNLINSOL_SUPERLUMT	Libraries	libsundials_sunlinsolsuperlumt. <i>lib</i> libsundials_fsunlinsolsuperlumt.a
	Header files	sunlinsol/sunlinsol_superlumt.h
SUNNONLINSOL_NEWTON	Libraries	libsundials_sunnonlinsolnewton. <i>lib</i>
<i>continued on next page</i>		

<i>continued from last page</i>			
		libsundials_fsunnonlinsolnewton.a	
	Header files	sunnonlinsol/sunnonlinsol_newton.h	
SUNNONLINSOL_FIXEDPOINT	Libraries	libsundials_sunnonlinsolfixedpoint. <i>lib</i> libsundials_fsunnonlinsolfixedpoint.a	
	Header files	sunnonlinsol/sunnonlinsol_fixedpoint.h	
CVODE	Libraries	libsundials_cvode. <i>lib</i>	libsundials_fcvcde.a
	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
CVODES	Header files	cvode/cvode_bbdpre.h	
		libsundials_cvodes. <i>lib</i>	
		cvodes/cvodes.h	cvodes/cvodes_impl.h
		cvodes/cvodes_direct.h	cvodes/cvodes_spils.h
ARKODE	Header files	cvodes/cvodes_bandpre.h	cvodes/cvodes_bbdpre.h
		libsundials_arkode. <i>lib</i>	libsundials_farkode.a
		arkode/arkode.h	arkode/arkode_impl.h
		arkode/arkode_ls.h	arkode/arkode_bandpre.h
IDA	Header files	arkode/arkode_bbdpre.h	
		libsundials_ida. <i>lib</i>	libsundials_fida.a
		ida/ida.h	ida/ida_impl.h
		ida/ida_direct.h	ida/ida_spils.h
IDAS	Header files	ida/ida_bbdpre.h	
		libsundials_idas. <i>lib</i>	
		idas/idas.h	idas/idas_impl.h
		idas/idas_direct.h	idas/idas_spils.h
KINSOL	Header files	idas/idas_bbdpre.h	
		libsundials_kinsol. <i>lib</i>	libsundials_fkinsol.a
		kinsol/kinsol.h	kinsol/kinsol_impl.h
		kinsol/kinsol_direct.h	kinsol/kinsol_ls.h
	Header files	kinsol/kinsol_spils.h	kinsol/kinsol_bbdpre.h

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.
Iterative 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 <code>mxstep</code> 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.

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 unable to recover.
CV_MEM_FAIL	-20	A memory allocation failed.
CV_MEM_NULL	-21	The <code>cvode_mem</code> 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 <code>CVodeMalloc</code> .
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.

CVLS linear solver interface

CVLS_SUCCESS	0	Successful function return.
CVLS_MEM_NULL	-1	The <code>cvode_mem</code> argument was NULL.
CVLS_LMEM_NULL	-2	The CVLS linear solver has not been initialized.
CVLS_ILL_INPUT	-3	The CVLS solver is not compatible with the current NVECTOR module.
CVLS_MEM_FAIL	-4	A memory allocation request failed.
CVLS_PMEM_NULL	-5	The preconditioner module has not been initialized.
CVLS_JACFUNC_UNRECVR	-6	The Jacobian function failed in an unrecoverable manner.
CVLS_JACFUNC_RECVR	-7	The Jacobian function had a recoverable error.
CVLS_SUNMAT_FAIL	-8	An error occurred with the current SUNMATRIX module.
CVLS_SUNLS_FAIL	-9	An error occurred with the current SUNLINSOL module.

CVDIAG linear solver module

CVDIAG_SUCCESS	0	Successful function return.
CVDIAG_MEM_NULL	-1	The <code>cvode_mem</code> argument was NULL.
CVDIAG_LMEM_NULL	-2	The CVDIAG linear solver has not been initialized.
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.
CVDIAG_RHSFUNC_UNRECVR	-6	The right-hand side function failed in an unrecoverable manner.
CVDIAG_RHSFUNC_RECVR	-7	The right-hand side function had a recoverable error.

Bibliography

- [1] KLU Sparse Matrix Factorization Library. <http://faculty.cse.tamu.edu/davis/suitesparse.html>.
- [2] SuperLU_MT Threaded Sparse Matrix Factorization Library. <http://crd-legacy.lbl.gov/~xiaoye/-SuperLU/>.
- [3] D. G. Anderson. Iterative procedures for nonlinear integral equations. *J. Assoc. Comput. Machinery*, 12:547–560, 1965.
- [4] P. N. Brown. A local convergence theory for combined inexact-Newton/finite difference projection methods. *SIAM J. Numer. Anal.*, 24(2):407–434, 1987.
- [5] P. N. Brown, G. D. Byrne, and A. C. Hindmarsh. VODE, a Variable-Coefficient ODE Solver. *SIAM J. Sci. Stat. Comput.*, 10:1038–1051, 1989.
- [6] P. N. Brown and A. C. Hindmarsh. Reduced Storage Matrix Methods in Stiff ODE Systems. *J. Appl. Math. & Comp.*, 31:49–91, 1989.
- [7] P. N. Brown and Y. Saad. Hybrid Krylov Methods for Nonlinear Systems of Equations. *SIAM J. Sci. Stat. Comput.*, 11:450–481, 1990.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] S. D. Cohen and A. C. Hindmarsh. CVODE User Guide. Technical Report UCRL-MA-118618, LLNL, September 1994.
- [13] S. D. Cohen and A. C. Hindmarsh. CVODE, a Stiff/Nonstiff ODE Solver in C. *Computers in Physics*, 10(2):138–143, 1996.
- [14] 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.
- [15] R. S. Dembo, S. C. Eisenstat, and T. Steihaug. Inexact Newton Methods. *SIAM J. Numer. Anal.*, 19:400–408, 1982.
- [16] 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.

- [17] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM, Philadelphia, 1996.
- [18] H. Fang and Y. Saad. Two classes of secant methods for nonlinear acceleration. *Numer. Linear Algebra Appl.*, 16:197–221, 2009.
- [19] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14:470–482, 1993.
- [20] 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.
- [21] K. L. Hiebert and L. F. Shampine. Implicitly Defined Output Points for Solutions of ODEs. Technical Report SAND80-0180, Sandia National Laboratories, February 1980.
- [22] 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.
- [23] A. C. Hindmarsh. Avoiding BDF Stability Barriers in the MOL Solution of Advection-Dominated Problems. *Appl. Num. Math.*, 17:311–318, 1995.
- [24] A. C. Hindmarsh. The PODE and IDA Algorithms. Technical Report UCRL-ID-141558, LLNL, December 2000.
- [25] 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.
- [26] A. C. Hindmarsh, R. Serban, and A. Collier. Example Programs for IDA v4.0.0-dev.2. Technical Report UCRL-SM-208113, LLNL, 2018.
- [27] A. C. Hindmarsh, R. Serban, and D. R. Reynolds. Example Programs for CVODE v4.0.0-dev.2. Technical report, LLNL, 2018. UCRL-SM-208110.
- [28] A. C. Hindmarsh and A. G. Taylor. PODE and KINSOL: Parallel Software for Differential and Nonlinear Systems. Technical Report UCRL-ID-129739, LLNL, February 1998.
- [29] 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.
- [30] C. T. Kelley. *Iterative Methods for Solving Linear and Nonlinear Equations*. SIAM, Philadelphia, 1995.
- [31] X. S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.
- [32] 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.
- [33] 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.
- [34] Daniel R. Reynolds. Example Programs for ARKODE v3.0.0-dev.2. Technical report, Southern Methodist University, 2018.
- [35] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14(2):461–469, 1993.
- [36] 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.

-
- [37] 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.
 - [38] H. F. Walker and P. Ni. Anderson acceleration for fixed-point iterations. *SIAM Jour. Num. Anal.*, 49(4):1715–1735, 2011.

Index

- Adams method, [13](#)
- BDF method, [13](#)
- BIG_REAL, [26](#), [105](#)
- booleantype, [26](#)
- CONSTR_VEC, [92](#)
- CV_ADAMS, [32](#), [64](#)
- CV_BAD_DKY, [50](#)
- CV_BAD_K, [50](#)
- CV_BAD_T, [50](#)
- CV_BDF, [32](#), [64](#)
- CV_CONSTR_FAIL, [39](#)
- CV_CONV_FAILURE, [38](#)
- CV_ERR_FAILURE, [38](#)
- CV_FIRST_RHSFUNC_ERR, [66](#)
- CV_FIRST_RHSFUNC_FAIL, [39](#)
- CV_ILL_INPUT, [32](#), [33](#), [37](#), [38](#), [41–45](#), [49](#), [65](#)
- CV_LINIT_FAIL, [38](#)
- CV_LSETUP_FAIL, [38](#), [67](#), [76](#)
- CV_LSOLVE_FAIL, [38](#)
- CV_MEM_FAIL, [32](#), [44](#), [57](#), [58](#)
- CV_MEM_NULL, [32](#), [33](#), [37](#), [38](#), [41–45](#), [49](#), [50](#), [52–58](#), [65](#)
- CV_NO_MALLOC, [33](#), [34](#), [38](#), [65](#)
- CV_NORMAL, [38](#)
- CV_ONE_STEP, [38](#)
- CV_REPTD_RHSFUNC_ERR, [39](#)
- CV_RHSFUNC_FAIL, [39](#), [65](#)
- CV_ROOT_RETURN, [38](#)
- CV_RTFUNC_FAIL, [39](#), [67](#)
- CV_SUCCESS, [32](#), [33](#), [37–39](#), [41–45](#), [49](#), [50](#), [52–58](#), [65](#)
- CV_TOO_CLOSE, [38](#)
- CV_TOO_MUCH_ACC, [38](#)
- CV_TOO_MUCH_WORK, [38](#)
- CV_TSTOP_RETURN, [38](#)
- CV_UNREC_RHSFUNC_ERR, [39](#), [66](#)
- CV_WARNING, [66](#)
- CVBANDPRE preconditioner
 - description, [72](#)
 - optional output, [73–74](#)
 - usage, [72–73](#)
 - user-callable functions, [73](#)
- CVBandPrecGetNumRhsEvals, [74](#)
- CVBandPrecGetWorkSpace, [74](#)
- CVBandPrecInit, [73](#)
- CVBBDPRE preconditioner
 - description, [74–75](#)
 - optional output, [79](#)
 - usage, [76–77](#)
 - user-callable functions, [77–79](#)
 - user-supplied functions, [75–76](#)
- CVBBDPrecGetNumGfnEvals, [79](#)
- CVBBDPrecGetWorkSpace, [79](#)
- CVBBDPrecInit, [77](#)
- CVBBDPrecReInit, [78](#)
- CVDIAG linear solver
 - Jacobian approximation used by, [36](#)
 - selection of, [36](#)
- CVDIAG linear solver interface
 - memory requirements, [63](#)
 - optional output, [63–64](#)
 - use in FCVODE, [90](#)
- CVDIag, [30](#), [35](#), [36](#)
- CVDIAG_ILL_INPUT, [36](#)
- CVDIAG_LMEM_NULL, [63](#), [64](#)
- CVDIAG_MEM_FAIL, [36](#)
- CVDIAG_MEM_NULL, [36](#), [63](#), [64](#)
- CVDIAG_SUCCESS, [36](#), [63](#), [64](#)
- CVDIagGetLastFlag, [64](#)
- CVDIagGetNumRhsEvals, [63](#)
- CVDIagGetReturnFlagName, [64](#)
- CVDIagGetWorkSpace, [63](#)
- CVDlsGetLastFlag, [62](#)
- CVDlsGetNumJacEvals, [59](#)
- CVDlsGetNumRhsEvals, [60](#)
- CVDlsGetReturnFlagName, [63](#)
- CVDlsGetWorkspace, [59](#)
- CVDlsJacFn, [69](#)
- CVDlsSetJacFn, [46](#)
- CVDlsSetLinearSolver, [36](#)
- CVErrHandlerFn, [66](#)
- CVEwtFn, [66](#)
- CVLS generic linear solver
 - SUNLINSOL compatibility, [35](#)
- CVLS linear solver interface

- convergence test, 48
- Jacobian approximation used by, 46
- Jacobian-vector product approximation used by, 46
- memory requirements, 59
- optional input, 45–48
- optional output, 59–63
- preconditioner setup function, 47, 71
- preconditioner solve function, 47, 70
- CVLS_ILL_INPUT, 36, 48, 73, 78
- CVLS_JACFUNC_RECVR, 67
- CVLS_JACFUNC_UNRECVR, 67
- CVLS_LMEM_NULL, 46–48, 59–62, 73, 78, 79
- CVLS_MEM_FAIL, 36, 73, 78
- CVLS_MEM_NULL, 36, 46–48, 59–62
- CVLS_PMEM_NULL, 74, 79
- CVLS_SUCCESS, 35, 46–48, 59–62, 74
- CVLS_SUNLS_FAIL, 36, 47, 48
- CVLsJacFn, 67
- CVLsJacTimesSetupFn, 69
- CVLsJacTimesVecFn, 69
- CVLsPrecSetupFn, 71
- CVLsPrecSolveFn, 70
- CVODE, 1
 - motivation for writing in C, 2
 - package structure, 21
 - relationship to CVODE, PVODE, 1–2
 - relationship to VODE, VODPK, 1
- CVODE linear solver interfaces, 21–22
 - CVDIAG, 36
 - CVLS, 35
 - selecting one, 35
- CVODE linear solvers
 - header files, 27
 - implementation details, 22
 - NVECTOR compatibility, 25
 - selecting one, 35
- CVode, 30, 38
- cvode/cvode.h, 27
- cvode/cvode_diag.h, 28
- cvode/cvode_ls.h, 27
- CVodeCreate, 32
- CVodeFree, 31, 32
- CVodeGetActualInitStep, 55
- CVodeGetCurrentOrder, 54
- CVodeGetCurrentStep, 55
- CVodeGetCurrentTime, 55
- CVodeGetDky, 49
- CVodeGetErrWeights, 56
- CVodeGetEstLocalErrors, 56
- CVodeGetIntegratorStats, 56
- CVodeGetLastLinFlag, 62
- CVodeGetLastOrder, 54
- CVodeGetLastStep, 54
- CVodeGetLinReturnFlagName, 62
- CVodeGetLinWorkSpace, 59
- CVodeGetNonlinSolvStats, 57
- CVodeGetNumErrTestFails, 54
- CVodeGetNumGEvals, 58
- CVodeGetNumJacEvals, 59
- CVodeGetNumJtimesEvals, 62
- CVodeGetNumJTSetupEvals, 61
- CVodeGetNumLinConvFails, 60
- CVodeGetNumLinIters, 60
- CVodeGetNumLinRhsEvals, 60
- CVodeGetNumLinSolvSetups, 53
- CVodeGetNumNonlinSolvConvFails, 57
- CVodeGetNumNonlinSolvIters, 57
- CVodeGetNumPrecEvals, 61
- CVodeGetNumPrecSolves, 61
- CVodeGetNumRhsEvals, 53
- CVodeGetNumStabLimOrderReds, 55
- CVodeGetNumSteps, 53
- CVodeGetReturnFlagName, 58
- CVodeGetRootInfo, 58
- CVodeGetTolScaleFactor, 56
- CVodeGetWorkSpace, 52
- CVodeInit, 32, 64
- CVodeReInit, 64
- CVodeRootInit, 37
- CVodeSetConstraints, 45
- CVodeSetEpsLin, 48
- CVodeSetErrFile, 39, 41
- CVodeSetErrHandlerFn, 41
- CVodeSetInitStep, 43
- CVodeSetJacFn, 46
- CVodeSetLinearSolver, 30, 35, 67, 150, 207
- CVodeSetMaxConvFails, 44
- CVodeSetMaxErrTestFails, 44
- CVodeSetMaxHnilWarns, 42
- CVodeSetMaxNonlinIters, 44
- CVodeSetMaxNumSteps, 42
- CVodeSetMaxOrd, 41
- CVodeSetMaxStep, 43
- CVodeSetMaxStepsBetweenJac, 46
- CVodeSetMinStep, 43
- CVodeSetNoInactiveRootWarn, 49
- CVodeSetNonlinConvCoef, 45
- CVodeSetNonLinearSolver, 37
- CVodeSetNonlinearSolver, 30, 36
- CVodeSetPreconditioner, 47, 48
- CVodeSetRootDirection, 49
- CVodeSetStabLimDet, 42
- CVodeSetStopTime, 43
- CVodeSetUserData, 41
- CVodeSStolerances, 33
- CVodeSVtolerances, 33
- CVodeWFtolerances, 33

- CVRhsFn, 32, 65
- CVRootFn, 67
- CVSetJacTimes, 47
- CVSpilsGetLastFlag, 62
- CVSpilsGetNumConvFails, 60
- CVSpilsGetNumJtimesEvals, 62
- CVSpilsGetNumJTSetupEvals, 61
- CVSpilsGetNumLinIters, 60
- CVSpilsGetNumPrecEvals, 61
- CVSpilsGetNumPrecSolves, 61
- CVSpilsGetNumRhsEvals, 60
- CVSpilsGetReturnFlagName, 63
- CVSpilsGetWorkspace, 59
- CVSpilsJacTimesSetupFn, 70
- CVSpilsJacTimesVecFn, 69
- CVSpilsPrecSetupFn, 72
- CVSpilsPrecSolveFn, 71
- CVSpilsSetEpsLin, 48
- CVSpilsSetJacTimes, 47
- CVSpilsSetLinearSolver, 36
- CVSpilsSetPreconditioner, 48
- data types
 - Fortran, 81
- eh_data, 66
- error control
 - order selection, 16–17
 - step size selection, 16
- error messages, 39
 - redirecting, 39
 - user-defined handler, 41, 66
- FCVBANDSETJAC, 87
- FCVBBDINIT, 97
- FCVBBDOPT, 97
- FCVBBDREINIT, 97
- FCVBJAC, 87
- FCVBPINIT, 95
- FCVBPOPT, 96
- FCVCOMMFN, 98
- FCVDENSESETJAC, 87
- FCVDIAG, 90
- FCVDJAC, 86
- FCVDKY, 91
- FCVDLSINIT, 86
- FCVEWT, 86
- FCVEWTSET, 86
- FCVFREE, 91
- FCVFUN, 84
- FCVGETERRWEIGHTS, 92
- FCVGETESTLOCALERR, 94
- FCVGLOCFN, 98
- FCVJTIMES, 88, 98
- FCVJTSETUP, 88, 98
- FCVLSINIT, 86
- FCVLSSETJAC, 89, 95, 97
- FCVLSSETPREC, 90
- FCVMALLOC, 86
- FCVMALLOC, 85
- FCVODE, 91
- FCVODE interface module
 - interface to the CVBANDPRE module, 95–96
 - interface to the CVBBDPRE module, 96–98
 - optional input and output, 92
 - rootfinding, 94
 - usage, 83–91
 - user-callable functions, 82–83
 - user-supplied functions, 83
- FCVPSET, 89
- FCVPSOL, 89
- FCVREINIT, 91
- FCVSETIIN, 92
- FCVSETRIN, 92
- FCVSETVIN, 92
- FCVSPARSESETJAC, 88
- FCVSPILSETJAC, 89
- FCVSPILSETPREC, 90
- FCVSPILSINIT, 86
- FCVSPJAC, 88
- FSUNBANDLINSOLINIT, 164
- FSUNDENSELINSOLINIT, 162
- FSUNFIXEDPOINTINIT, 224
- FSUNKLUINIT, 172
- FSUNKLUREINIT, 172
- FSUNKLUSETORDERING, 173
- FSUNLAPACKBANDINIT, 169
- FSUNLAPACKDENSEINIT, 166
- FSUNMASSBANDLINSOLINIT, 165
- FSUNMASSDENSELINSOLINIT, 163
- FSUNMASSKLUINIT, 172
- FSUNMASSKLUREINIT, 173
- FSUNMASSKLUSEORDERING, 173
- FSUNMASSLAPACKBANDINIT, 169
- FSUNMASSLAPACKDENSEINIT, 167
- FSUNMASSPCGINIT, 204
- FSUNMASSPCGSETMAXL, 205
- FSUNMASSPCGSETPRECTYPE, 204
- FSUNMASSSPBCGSINIT, 193
- FSUNMASSSPBCGSSETMAXL, 194
- FSUNMASSSPBCGSSETPRECTYPE, 194
- FSUNMASSSPFGMRINIT, 188
- FSUNMASSSPFGMRSETGSTYPE, 188
- FSUNMASSSPFGMRSETMAXRS, 189
- FSUNMASSSPFGMRSETPRECTYPE, 189
- FSUNMASSSPGMRINIT, 182
- FSUNMASSSPGMRSETGSTYPE, 182
- FSUNMASSSPGMRSETMAXRS, 183
- FSUNMASSSPGMRSETPRECTYPE, 183

- FSUNMASSSPTFQMRINIT, 198
- FSUNMASSSPTFQMRSETMAXL, 199
- FSUNMASSSPTFQMRSETPRECTYPE, 199
- FSUNMASSSUPERLUMTINIT, 177
- FSUNMASSSUPERLUMTSETORDERING, 178
- FSUNNEWTONINIT, 220
- FSUNPCGINIT, 203
- FSUNPCGSETMAXL, 204
- FSUNPCGSETPRECTYPE, 204
- FSUNSPBCGSINIT, 193
- FSUNSPBCGSSETMAXL, 194
- FSUNSPBCGSSETPRECTYPE, 194
- FSUNSPFGMRINIT, 187
- FSUNSPFGMRSETGSTYPE, 188
- FSUNSPFGMRSETMAXRS, 189
- FSUNSPFGMRSETPRECTYPE, 188
- FSUNSPGMRINIT, 181
- FSUNSPGMRSETGSTYPE, 182
- FSUNSPGMRSETMAXRS, 183
- FSUNSPGMRSETPRECTYPE, 182
- FSUNSPTFQMRINIT, 198
- FSUNSPTFQMRSETMAXL, 199
- FSUNSPTFQMRSETPRECTYPE, 198
- FSUNSUPERLUMTINIT, 177
- FSUNSUPERLUMTSETORDERING, 177

- getDevData(N_Vector v), 123, 125
- getGlobalSize(N_Vector v), 123, 126
- getHostData(N_Vector v), 123, 125
- getMPIComm(N_Vector v), 123, 126
- getSize(N_Vector v), 123, 125

- half-bandwidths, 73, 77
- header files, 27, 72, 76
- HNIL_WARNINGS, 92

- INIT_STEP, 92
- IOUT, 92, 93
- itask, 30, 38

- Jacobian approximation function
 - band
 - use in FCVODE, 87
 - dense
 - use in FCVODE, 86
 - diagonal
 - difference quotient, 36
 - difference quotient, 46
 - Jacobian times vector
 - difference quotient, 47
 - use in FCVODE, 88
 - user-supplied, 47
 - Jacobian-vector product
 - user-supplied, 69
 - Jacobian-vector setup, 69–70
 - sparse
 - use in FCVODE, 88
 - user-supplied, 46, 67–69
 - Jacobian update frequency
 - optional input, 45
 - Jacobian-vector product
 - setup and solve phases, 22

- lmm, 32, 64
- LSODE, 1

- MAX_CONVFAIL, 92
- MAX_ERRFAIL, 92
- MAX_NITERS, 92
- MAX_NSTEPS, 92
- MAX_ORD, 92
- MAX_STEP, 92
- maxord, 42, 64
- memory requirements
 - CVBANDPRE preconditioner, 74
 - CVBBDPRE preconditioner, 79
 - CVDIAG linear solver interface, 63
 - CVLS linear solver interface, 59
 - CVODE solver, 52
- MIN_STEP, 92

- N_VCloneVectorArray, 100
- N_VCloneVectorArray_OpenMP, 115
- N_VCloneVectorArray_Parallel, 113
- N_VCloneVectorArray_ParHyp, 119
- N_VCloneVectorArray_Petsc, 121
- N_VCloneVectorArray_Pthreads, 118
- N_VCloneVectorArray_Serial, 110
- N_VCloneVectorArrayEmpty, 100
- N_VCloneVectorArrayEmpty_OpenMP, 115
- N_VCloneVectorArrayEmpty_Parallel, 113
- N_VCloneVectorArrayEmpty_ParHyp, 120
- N_VCloneVectorArrayEmpty_Petsc, 121
- N_VCloneVectorArrayEmpty_Pthreads, 118
- N_VCloneVectorArrayEmpty_Serial, 110
- N_VCopyFromDevice_Cuda, 124
- N_VCopyFromDevice_Raja, 127
- N_VCopyToDevice_Cuda, 124
- N_VCopyToDevice_Raja, 127
- N_VDestroyVectorArray, 100
- N_VDestroyVectorArray_OpenMP, 115
- N_VDestroyVectorArray_Parallel, 113
- N_VDestroyVectorArray_ParHyp, 120
- N_VDestroyVectorArray_Petsc, 121
- N_VDestroyVectorArray_Pthreads, 118
- N_VDestroyVectorArray_Serial, 110
- N_Vector, 27, 99
- N_VGetDeviceArrayPointer_Cuda, 124
- N_VGetDeviceArrayPointer_Raja, 127
- N_VGetHostArrayPointer_Cuda, 124

- N_VGetHostArrayPointer_Raja, 127
- N_VGetLength_Cuda, 124
- N_VGetLength_OpenMP, 116
- N_VGetLength_Parallel, 113
- N_VGetLength_Pthreads, 118
- N_VGetLength_Raja, 127
- N_VGetLength_Serial, 110
- N_VGetLocalLength_Parallel, 113
- N_VGetVector_ParHyp, 119
- N_VGetVector_Petsc, 121
- N_VMake_Cuda, 124
- N_VMake_OpenMP, 115
- N_VMake_Parallel, 112
- N_VMake_ParHyp, 119
- N_VMake_Petsc, 121
- N_VMake_Pthreads, 118
- N_VMake_Raja, 126
- N_VMake_Serial, 110
- N_VNew_Cuda, 123
- N_VNew_OpenMP, 115
- N_VNew_Parallel, 112
- N_VNew_Pthreads, 117
- N_VNew_Raja, 126
- N_VNew_SensWrapper, 217
- N_VNew_Serial, 110
- N_VNewEmpty_Cuda, 124
- N_VNewEmpty_OpenMP, 115
- N_VNewEmpty_Parallel, 112
- N_VNewEmpty_ParHyp, 119
- N_VNewEmpty_Petsc, 121
- N_VNewEmpty_Pthreads, 117
- N_VNewEmpty_Raja, 126
- N_VNewEmpty_SensWrapper, 217
- N_VNewEmpty_Serial, 110
- N_VPrint_Cuda, 124
- N_VPrint_OpenMP, 116
- N_VPrint_Parallel, 113
- N_VPrint_ParHyp, 120
- N_VPrint_Petsc, 121
- N_VPrint_Pthreads, 118
- N_VPrint_Raja, 127
- N_VPrint_Serial, 110
- N_VPrintFile_Cuda, 124
- N_VPrintFile_OpenMP, 116
- N_VPrintFile_Parallel, 113
- N_VPrintFile_ParHyp, 120
- N_VPrintFile_Petsc, 121
- N_VPrintFile_Pthreads, 118
- N_VPrintFile_Raja, 127
- N_VPrintFile_Serial, 110
- NLCONV_COEF, 92
- nonlinear system
 - Convergence test, 15
 - definition, 13–14
 - Newton iteration, 14–15
- NV_COMM_P, 112
- NV_CONTENT_OMP, 114
- NV_CONTENT_P, 111
- NV_CONTENT_PT, 117
- NV_CONTENT_S, 109
- NV_DATA_OMP, 114
- NV_DATA_P, 111
- NV_DATA_PT, 117
- NV_DATA_S, 109
- NV_GLOBLLENGTH_P, 111
- NV_Ith_OMP, 115
- NV_Ith_P, 112
- NV_Ith_PT, 117
- NV_Ith_S, 109
- NV_LENGTH_OMP, 114
- NV_LENGTH_PT, 117
- NV_LENGTH_S, 109
- NV_LOCLENGTH_P, 111
- NV_NUM_THREADS_OMP, 114
- NV_NUM_THREADS_PT, 117
- NV_OWN_DATA_OMP, 114
- NV_OWN_DATA_P, 111
- NV_OWN_DATA_PT, 117
- NV_OWN_DATA_S, 109
- NVECTOR module, 99
- optional input
 - generic linear solver interface, 45–48
 - iterative linear solver, 47–48
 - matrix-based linear solver, 46
 - matrix-free linear solver, 46–47
 - rootfinding, 48–49
 - solver, 39–45
- optional output
 - band-block-diagonal preconditioner, 79
 - banded preconditioner, 73–74
 - diagonal linear solver interface, 63–64
 - generic linear solver interface, 59–63
 - interpolated solution, 49
 - solver, 52–58
 - version, 50–52
- output mode, 17, 38
- portability, 26
 - Fortran, 81
- Preconditioner setup routine
 - use in FCVODE, 89
- Preconditioner solve routine
 - use in FCVODE, 89
- Preconditioner update frequency
 - optional input, 45
- preconditioning
 - advice on, 17, 22

- band-block diagonal, 74
- banded, 72
- setup and solve phases, 22
- user-supplied, 47–48, 70, 71
- PVODE, 1
- RCONST, 26
- realtype, 26
- reinitialization, 64
- right-hand side function, 65
- Rootfinding, 19, 30, 37, 94
- ROUT, 92, 93
- SM_COLS_B, 141
- SM_COLS_D, 137
- SM_COLUMN_B, 68, 141
- SM_COLUMN_D, 68, 137
- SM_COLUMN_ELEMENT_B, 68, 141
- SM_COLUMNS_B, 141
- SM_COLUMNS_D, 136
- SM_COLUMNS_S, 147
- SM_CONTENT_B, 141
- SM_CONTENT_D, 136
- SM_CONTENT_S, 145
- SM_DATA_B, 141
- SM_DATA_D, 137
- SM_DATA_S, 147
- SM_ELEMENT_B, 68, 141
- SM_ELEMENT_D, 68, 137
- SM_INDEXPTRS_S, 147
- SM_INDEXVALS_S, 147
- SM_LBAND_B, 141
- SM_LDATA_B, 141
- SM_LDATA_D, 136
- SM_LDIM_B, 141
- SM_NNZ_S, 69, 147
- SM_NP_S, 147
- SM_ROWS_B, 141
- SM_ROWS_D, 136
- SM_ROWS_S, 147
- SM_SPARSETYPE_S, 147
- SM_SUBAND_B, 141
- SM_UBAND_B, 141
- SMALL_REAL, 26
- STAB_LIM, 92
- Stability limit detection, 18
- step size bounds, 43
- STOP_TIME, 92
- SUNBandLinearSolver, 164
- SUNBandMatrix, 29, 142
- SUNBandMatrix_Cols, 143
- SUNBandMatrix_Column, 143
- SUNBandMatrix_Columns, 142
- SUNBandMatrix_Data, 143
- SUNBandMatrix_LDim, 143
- SUNBandMatrix_LowerBandwidth, 142
- SUNBandMatrix_Print, 142
- SUNBandMatrix_Rows, 142
- SUNBandMatrix_StoredUpperBandwidth, 142
- SUNBandMatrix_UpperBandwidth, 142
- SUNDenseLinearSolver, 162
- SUNDenseMatrix, 29, 137
- SUNDenseMatrix_Cols, 138
- SUNDenseMatrix_Column, 138
- SUNDenseMatrix_Columns, 138
- SUNDenseMatrix_Data, 138
- SUNDenseMatrix_LData, 138
- SUNDenseMatrix_Print, 137
- SUNDenseMatrix_Rows, 138
- sundials/sundials_linearsolver.h, 153
- sundials_nonlinearsolver.h, 27
- sundials_nvector.h, 27
- sundials_types.h, 26, 27
- SUNDIALSGetVersion, 50
- SUNDIALSGetVersionNumber, 52
- sunindextype, 26
- SUNKLU, 172
- SUNKLUReInit, 172
- SUNKLUSetOrdering, 172
- SUNLapackBand, 168
- SUNLapackDense, 166
- SUNLineaerSolver, 153
- SUNLinearSolver, 35, 160
- SUNLinearSolver module, 153
- sunlinsol/sunlinsol_band.h, 27
- sunlinsol/sunlinsol_dense.h, 27
- sunlinsol/sunlinsol_klu.h, 27
- sunlinsol/sunlinsol_lapackband.h, 27
- sunlinsol/sunlinsol_lapackdense.h, 27
- sunlinsol/sunlinsol_pcg.h, 28
- sunlinsol/sunlinsol_spgmrs.h, 27
- sunlinsol/sunlinsol_spgmrs.h, 27
- sunlinsol/sunlinsol_sptfqmr.h, 28
- sunlinsol/sunlinsol_superlumt.h, 27
- SUNLinSol_Band, 35, 164
- SUNLinSol_Dense, 35, 162
- SUNLinSol_KLU, 35, 170
- SUNLinSol_KLUReInit, 171
- SUNLinSol_KLUSetOrdering, 172
- SUNLinSol_LapackBand, 35, 168
- SUNLinSol_LapackDense, 35, 166
- SUNLinSol_PCG, 35, 202–204
- SUNLinSol_PCGSetMax1, 203
- SUNLinSol_PCGSetPrecType, 203
- SUNLinSol_SPBCGS, 35, 192, 193
- SUNLinSol_SPBCGSSetMax1, 192, 193
- SUNLinSol_SPBCGSSetPrecType, 192, 193

- SUNLinSol_SPGFMR, 35, 186–188
- SUNLinSol_SPGFMRSetGSType, 187
- SUNLinSol_SPGFMRSetMaxRestarts, 187
- SUNLinSol_SPGFMRSetPrecType, 186, 187
- SUNLinSol_SPGMR, 35, 180–182
- SUNLinSol_SPGMRSetGSType, 181
- SUNLinSol_SPGMRSetMaxRestarts, 181
- SUNLinSol_SPGMRSetPrecType, 180, 181
- SUNLinSol_SPTFQMR, 35, 197, 198
- SUNLinSol_SPTFQMRSetMaxl, 197, 198
- SUNLinSol_SPTFQMRSetPrecType, 197, 198
- SUNLinSol_SuperLUMT, 35, 176
- SUNLinSol_SuperLUMTSetOrdering, 176, 177
- SUNLinSolFree, 31, 154, 155
- SUNLinSolGetType, 154, 207
- SUNLinSolInitialize, 154
- SUNLinSolLastFlag, 157
- SUNLinSolNumIters, 157
- SUNLinSolResNorm, 157
- SUNLinSolSetATimes, 156
- SUNLinSolSetPreconditioner, 156
- SUNLinSolSetScalingVectors, 156
- SUNLinSolSetup, 154, 155
- SUNLinSolSolve, 154, 155
- SUNLinSolSpace, 157
- SUNMatDestroy, 31
- SUNMatrix, 133
- SUNMatrix module, 133
- SUNNonlinearSolver, 27, 209
- SUNNonlinearSolver module, 209
- SUNNONLINEARSOLVER_DIRECT, 154
- SUNNONLINEARSOLVER_FIXEDPOINT, 210
- SUNNONLINEARSOLVER_ITERATIVE, 154
- SUNNONLINEARSOLVER_ROOTFIND, 210
- SUNNonlinSol_FixedPoint, 222, 223
- SUNNonlinSol_FixedPointSens, 222
- SUNNonlinSol_Newton, 219
- SUNNonlinSol_NewtonSens, 219
- SUNNonlinSolFree, 31, 211
- SUNNonlinSolGetCurIter, 213
- SUNNonlinSolGetNumIters, 213
- SUNNonlinSolGetSysFn_FixedPoint, 222
- SUNNonlinSolGetSysFn_Newton, 219
- SUNNonlinSolGetType, 210
- SUNNonlinSolInitialize, 210
- SUNNonlinSolLSetupFn, 211
- SUNNonlinSolSetConvTestFn, 212
- SUNNonlinSolSetLSolveFn, 212
- SUNNonlinSolSetMaxIters, 212
- SUNNonlinSolSetSysFn, 211
- SUNNonlinSolSetup, 210
- SUNNonlinSolSolve, 210
- SUNPCG, 203
- SUNPCGSetMaxl, 203
- SUNPCGSetPrecType, 203
- SUNSparseFromBandMatrix, 148
- SUNSparseFromDenseMatrix, 147
- SUNSparseMatrix, 29, 147
- SUNSparseMatrix_Columns, 148
- SUNSparseMatrix_Data, 149
- SUNSparseMatrix_IndexPointers, 149
- SUNSparseMatrix_IndexValues, 149
- SUNSparseMatrix_NNZ, 69, 149
- SUNSparseMatrix_NP, 149
- SUNSparseMatrix_Print, 148
- SUNSparseMatrix_Realloc, 148
- SUNSparseMatrix_Reallocate, 148
- SUNSparseMatrix_Rows, 148
- SUNSparseMatrix_SparseType, 149
- SUNSPBCGS, 193
- SUNSPBCGSSetMaxl, 193
- SUNSPBCGSSetPrecType, 193
- SUNSPFGMR, 187
- SUNSPFGMRSetGSType, 187
- SUNSPFGMRSetMaxRestarts, 187
- SUNSPFGMRSetPrecType, 187
- SUNSPGMR, 181
- SUNSPGMRSetGSType, 181
- SUNSPGMRSetMaxRestarts, 181
- SUNSPGMRSetPrecType, 181
- SUNSPTFQMR, 198
- SUNSPTFQMRSetMaxl, 198
- SUNSPTFQMRSetPrecType, 198
- SUNSuperLUMT, 176
- SUNSuperLUMTSetOrdering, 176
- tolerances, 14, 34, 66
- UNIT_ROUNDOFF, 26
- User main program
 - CVBANDPRE usage, 72
 - CVBBDPRE usage, 76
 - FCVBBD usage, 96
 - FCVBP usage, 95
 - FCVODE usage, 83
 - IVP solution, 28
- user_data, 41, 65–67, 76
- VODE, 1
- VODPK, 1
- weighted root-mean-square norm, 14

