User Documentation for CVODES v3.2.0 (SUNDIALS v3.2.0)

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

September 28, 2018

UCRL-SM-208111

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.

Approved for public release; further dissemination unlimited

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 . . . 10
  - 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
- 2.5 Pure quadrature integration . . . 20
- 2.6 Forward sensitivity analysis . . . 20
  - 2.6.1 Forward sensitivity methods . . . 21
  - 2.6.2 Selection of the absolute tolerances for sensitivity variables . . . 22
  - 2.6.3 Evaluation of the sensitivity right-hand side . . . 22
  - 2.6.4 Quadratures depending on forward sensitivities . . . 23
- 2.7 Adjoint sensitivity analysis . . . 23
  - 2.7.1 Checkpointing scheme . . . 25
- 2.8 Second-order sensitivity analysis . . . 26

3 Code Organization 29
- 3.1 SUNDIALS organization . . . 29
- 3.2 CVODES organization . . . 29

4 Using CVODES for IVP Solution 35
- 4.1 Access to library and header files . . . 35
- 4.2 Data Types . . . 36
  - 4.2.1 Floating point types . . . 36
  - 4.2.2 Integer types used for vector and matrix indices . . . 36
- 4.3 Header files . . . 37
- 4.4 A skeleton of the user's main program . . . 38
- 4.5 User-callable functions . . . 40
  - 4.5.1 CVODES initialization and deallocation functions . . . 40
  - 4.5.2 CVODES tolerance specification functions . . . 42

4.5.3 Linear solver interface functions . . . 44
4.5.4 Rootfinding initialization function . . . 46
4.5.5 CVODES solver function . . . 46
4.5.6 Optional input functions . . . 48
4.5.6.1 Main solver optional input functions . . . 48
4.5.6.2 Direct linear solver interface optional input functions . . . 54
4.5.6.3 Iterative linear solver interface optional input functions . . . 54
4.5.6.4 Rootfinding optional input functions . . . 56
4.5.7 Interpolated output function . . . 57
4.5.8 Optional output functions . . . 57
4.5.8.1 SUNDIALS version information . . . 57
4.5.8.2 Main solver optional output functions . . . 59
4.5.8.3 Rootfinding optional output functions . . . 65
4.5.8.4 Direct linear solver interface optional output functions . . . 66
4.5.8.5 Iterative linear solver interface optional output functions . . . 67
4.5.8.6 Diagonal linear solver interface optional output functions . . . 70
4.5.9 CVODES reinitialization function . . . 72
4.6 User-supplied functions . . . 72
4.6.1 ODE right-hand side . . . 73
4.6.2 Error message handler function . . . 73
4.6.3 Error weight function . . . 74
4.6.4 Rootfinding function . . . 74
4.6.5 Jacobian information (direct method Jacobian) . . . 75
4.6.6 Jacobian information (matrix-vector product) . . . 76
4.6.7 Jacobian information (matrix-vector setup) . . . 77
4.6.8 Preconditioning (linear system solution) . . . 77
4.6.9 Preconditioning (Jacobian data) . . . 78
4.7 Integration of pure quadrature equations . . . 79
4.7.1 Quadrature initialization and deallocation functions . . . 80
4.7.2 CVODES solver function . . . 81
4.7.3 Quadrature extraction functions . . . 82
4.7.4 Optional inputs for quadrature integration . . . 83
4.7.5 Optional outputs for quadrature integration . . . 84
4.7.6 User-supplied function for quadrature integration . . . 85
4.8 Preconditioner modules . . . 86
4.8.1 A serial banded preconditioner module . . . 86
4.8.2 A parallel band-block-diagonal preconditioner module . . . 88

5 Using CVODES for Forward Sensitivity Analysis 95
5.1 A skeleton of the user's main program . . . 95
5.2 User-callable routines for forward sensitivity analysis . . . 97
5.2.1 Forward sensitivity initialization and deallocation functions . . . 98
5.2.2 Forward sensitivity tolerance specification functions . . . 101
5.2.3 CVODES solver function . . . 102
5.2.4 Forward sensitivity extraction functions . . . 102
5.2.5 Optional inputs for forward sensitivity analysis . . . 104
5.2.6 Optional outputs for forward sensitivity analysis . . . 105
5.3 User-supplied routines for forward sensitivity analysis . . . 110
5.3.1 Sensitivity equations right-hand side (all at once) . . . 110
5.3.2 Sensitivity equations right-hand side (one at a time) . . . 111
5.4 Integration of quadrature equations depending on forward sensitivities . . . 111
5.4.1 Sensitivity-dependent quadrature initialization and deallocation . . . 113
5.4.2 CVODES solver function . . . 114
5.4.3 Sensitivity-dependent quadrature extraction functions . . . 115

5.4.4 Optional inputs for sensitivity-dependent quadrature integration 116
5.4.5 Optional outputs for sensitivity-dependent quadrature integration 118
5.4.6 User-supplied function for sensitivity-dependent quadrature integration 119
5.5 Note on using partial error control 120

6 Using CVODES for Adjoint Sensitivity Analysis 123
6.1 A skeleton of the user's main program 123
6.2 User-callable functions for adjoint sensitivity analysis 126
6.2.1 Adjoint sensitivity allocation and deallocation functions 126
6.2.2 Forward integration function 127
6.2.3 Backward problem initialization functions 128
6.2.4 Tolerance specification functions for backward problem 131
6.2.5 Linear solver initialization functions for backward problem 131
6.2.6 Backward integration function 133
6.2.7 Adjoint sensitivity optional input 134
6.2.8 Optional input functions for the backward problem 135
6.2.8.1 Main solver optional input functions 135
6.2.8.2 Direct linear solver interface optional input functions 135
6.2.8.3 SPILS linear solvers 136
6.2.9 Optional output functions for the backward problem 138
6.2.10 Backward integration of quadrature equations 139
6.2.10.1 Backward quadrature initialization functions 139
6.2.10.2 Backward quadrature extraction function 141
6.2.10.3 Optional input/output functions for backward quadrature integration 141
6.3 User-supplied functions for adjoint sensitivity analysis 141
6.3.1 ODE right-hand side for the backward problem 142
6.3.2 ODE right-hand side for the backward problem depending on the forward sensitivities 142
6.3.3 Quadrature right-hand side for the backward problem 143
6.3.4 Sensitivity-dependent quadrature right-hand side for the backward problem . . 144
6.3.5 Jacobian information for the backward problem (direct method Jacobian) . . . 144
6.3.6 Jacobian information for the backward problem (matrix-vector product) 146
6.3.7 Jacobian information for the backward problem (matrix-vector setup) 147
6.3.8 Preconditioning for the backward problem (linear system solution) 149
6.3.9 Preconditioning for the backward problem (Jacobian data) 150
6.4 Using CVODES preconditioner modules for the backward problem 151
6.4.1 Using the banded preconditioner CVBANDPRE 151
6.4.2 Using the band-block-diagonal preconditioner CVBBDPRE 152
6.4.2.1 Initialization of CVBBDPRE 152
6.4.2.2 User-supplied functions for CVBBDPRE 153

7 Description of the NVECTOR module 155
7.1 The NVECTOR_SERIAL implementation 160
7.2 The NVECTOR_PARALLEL implementation 162
7.3 The NVECTOR_OPENMP implementation 165
7.4 The NVECTOR_PTHREADS implementation 167
7.5 The NVECTOR_PARHYP implementation 170
7.6 The NVECTOR_PETSC implementation 171
7.7 The NVECTOR_CUDA implementation 173
7.8 The NVECTOR_RAJA implementation 176
7.9 NVECTOR Examples 178
7.10 NVECTOR functions used by CVODES 180

8 Description of the SUNMatrix module **183**
8.1 The SUNMatrix_Dense implementation . . . 186
8.2 The SUNMatrix_Band implementation . . . 189
8.3 The SUNMatrix_Sparse implementation . . . 193
8.4 SUNMatrix Examples . . . 199
8.5 SUNMatrix functions used by CVODES . . . 200

9 Description of the SUNLinearSolver module **203**
9.1 Description of the client-supplied SUNLinearSolver routines . . . 208
9.2 Compatibility of SUNLinearSolver modules . . . 209
9.3 The SUNLinearSolver_Dense implementation . . . 210
9.4 The SUNLinearSolver_Band implementation . . . 212
9.5 The SUNLinearSolver_LapackDense implementation . . . 213
9.6 The SUNLinearSolver_LapackBand implementation . . . 215
9.7 The SUNLinearSolver_KLU implementation . . . 216
9.8 The SUNLinearSolver_SuperLUMT implementation . . . 219
9.9 The SUNLinearSolver_SPGMR implementation . . . 222
9.10 The SUNLinearSolver_SPFGMR implementation . . . 225
9.11 The SUNLinearSolver_SPBCGS implementation . . . 229
9.12 The SUNLinearSolver_SPTFQMR implementation . . . 232
9.13 The SUNLinearSolver_PCG implementation . . . 235
9.14 SUNLinearSolver Examples . . . 238
9.15 SUNLinearSolver functions used by CVODES . . . 239

A SUNDIALS Package Installation Procedure **241**
A.1 CMake-based installation . . . 242
A.1.1 Configuring, building, and installing on Unix-like systems . . . 242
A.1.2 Configuration options (Unix/Linux) . . . 244
A.1.3 Configuration examples . . . 250
A.1.4 Working with external Libraries . . . 251
A.1.5 Testing the build and installation . . . 253
A.2 Building and Running Examples . . . 253
A.3 Configuring, building, and installing on Windows . . . 254
A.4 Installed libraries and exported header files . . . 254

B CVODES Constants **259**
B.1 CVODES input constants . . . 259
B.2 CVODES output constants . . . 260

Bibliography **263**

Index **265**

List of Tables

4.1 SUNDIALS linear solver interfaces and vector implementations that can be used for each. 41
4.2 Optional inputs for CVODES, CVDLS, and CVSPILS 48
4.3 Optional outputs from CVODES, CVDLS, CVDIAG, and CVSPILS 58

5.1 Forward sensitivity optional inputs 104
5.2 Forward sensitivity optional outputs 106

7.1 Vector Identifications associated with vector kernels supplied with SUNDIALS. 157
7.2 Description of the NVECTOR operations 157
7.3 List of vector functions usage by CVODES code modules 181

8.1 Identifiers associated with matrix kernels supplied with SUNDIALS. 184
8.2 Description of the `SUNMatrix` operations 184
8.3 SUNDIALS matrix interfaces and vector implementations that can be used for each. . . 185
8.4 List of matrix functions usage by CVODES code modules 200

9.1 Identifiers associated with linear solver kernels supplied with SUNDIALS. 205
9.2 Description of the `SUNLinearSolver` operations 205
9.3 SUNDIALS direct linear solvers and matrix implementations that can be used for each. 209
9.4 Description of the `SUNLinearSolver` error codes 210
9.5 List of linear solver functions usage by CVODES code modules 239

A.1 SUNDIALS libraries and header files 254

List of Figures

2.1 Illustration of the checkpointing algorithm for generation of the forward solution during the integration of the adjoint system. 25

3.1 High-level diagram of the SUNDIALS suite 30
3.2 Organization of the SUNDIALS suite 31
3.3 Overall structure of the CVODES package 32

8.1 Diagram of the storage for a SUNMATRIX_BAND object 190
8.2 Diagram of the storage for a compressed-sparse-column matrix 196

A.1 Initial *ccmake* configuration screen 243
A.2 Changing the *instdir* 244

Chapter 1

Introduction

CVODES [34] is part of a software family called SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic equation Solvers [20]. This suite consists of CVODE, ARKODE, KINSOL and IDA, and variants of these with sensitivity analysis capabilities. CVODES is a solver for stiff and nonstiff initial value problems (IVPs) for systems of ordinary differential equation (ODEs). In addition to solving stiff and nonstiff ODE systems, CVODES has sensitivity analysis capabilities, using either the forward or the adjoint methods.

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 [3] and VODPK [5]. 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 [30]. 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 [4]. The capabilities of both VODE and VODPK have been combined in the C-language package CVODE [10].

At present, CVODE may utilize a variety of Krylov methods provided in SUNDIALS that can be used in conjuction with Newton iteration: these include the GMRES (Generalized Minimal RESidual) [33], FGMRES (Flexible Generalized Minimum RESidual) [32], Bi-CGStab (Bi-Conjugate Gradient Stabilized) [36], TFQMR (Transpose-Free Quasi-Minimal Residual) [14], and PCG (Preconditioned Conjugate Gradient) [15] linear iterative methods. As Krylov methods, these require almost no matrix storage for solving the Newton equations as compared to direct methods. However, the algorithms allow for a user-supplied preconditioner matrix, and for most problems preconditioning is essential for an efficient solution. For very large stiff ODE systems, the Krylov methods are preferable over direct linear solver methods, and are often the only feasible choice. Among the Krylov methods in SUNDIALS, we recommend GMRES as the best overall choice. However, users are encouraged to compare all options, especially if encountering convergence failures with GMRES. Bi-CGStab and TFQMR have an advantage in storage requirements, in that the number of workspace vectors they require is fixed, while that number for GMRES depends on the desired Krylov subspace size. FGMRES has an advantage in that it is designed to support preconditioners that vary between iterations (e.g. iterative methods). PCG exhibits rapid convergence and minimal workspace vectors, but only works for symmetric linear systems.

In the process of translating the VODE and VODPK algorithms into C, the overall CVODE organization has been changed considerably. One key feature of the CVODE organization is that the linear system solvers comprise a layer of code modules that is separated from the integration algorithm, allowing for easy modification and expansion of the linear solver array. A second key feature is a

separate module devoted to vector operations; this facilitated the extension to multiprosessor environments with minimal impacts on the rest of the solver, resulting in PVODE [7], the parallel variant of CVODE.

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

Development of CVODES 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 functions 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 to be linked into an executable file. SUNDIALS (and thus CVODES) is supplied with serial, MPI-parallel, and both openMP and Pthreads thread-parallel NVECTOR implementations.

There were several motivations for choosing the C language for CVODE, and later for CVODES. 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. Finally, we prefer C over C++ for CVODES 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 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 §4.5.6.1 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 thread-safety issue when using ajdoint sensitivity analysis.

Fixed a problem with setting `sunindextype` which would occur with some compilers (e.g. armclang) that did not define `__STDC_VERSION__`.

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

Changed the name of the RAJA NVECTOR library to `libsundials_nveccudaraja.lib` from `libsundials_nvecraja.lib` to better reflect that we only support CUDA as a backend for RAJA currently.

Several changes were made to the build system:

- CMake 3.1.3 is now the minimum required CMake version.
- Deprecate the behavior of the `SUNDIALS_INDEX_TYPE` CMake option and added the `SUNDIALS_INDEX_SIZE` CMake option to select the `sunindextype` integer size.
- The native CMake FindMPI module is now used to locate an MPI installation.

- If MPI is enabled and MPI compiler wrappers are not set, the build system will check if CMAKE_<language>_COMPILER can compile MPI programs before trying to locate and use an MPI installation.
- The previous options for setting MPI compiler wrappers and the executable for running MPI programs have been have been depreated. The new options that align with those used in native CMake FindMPI module are MPI_C_COMPILER, MPI_CXX_COMPILER, MPI_Fortran_COMPILER, and MPIEXEC_EXECUTABLE.
- When a Fortran name-mangling scheme is needed (e.g., 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 new example, cvRoberts_FSA_dns_Switch.c, which demonstrates switching on/off forward sensitivity computations. This example came from the usage notes page of the SUNDIALS website.
- The misnamed function CVSpilsSetJacTimesSetupFnBS has been deprecated and replaced by CVSpilsSetJacTimesBS. The deprecated function CVSpilsSetJacTimesSetupFnBS will be removed in the next major release.
- 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.

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 th web site, https://software.llnl.gov/RAJA/. These additions are accompanied by additions to various interface functions and to user documentation.

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

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

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

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

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

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

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

A bug fix was made in `CVodeFree` to call `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.

A bug was fixed in the interpolation functions used in solving backward problems for adjoint sensitivity analysis.

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

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

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.

In interpolation routines for backward problems, added logic to bypass sensitivity interpolation if input sensitivity argument is NULL.

New examples were added for use of sparse direct solvers within sensitivity integrations and for use of openMP.

Minor corrections and additions were made to the CVODES solver, 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 CVODES 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 CVODES.

Otherwise, only relatively minor modifications were made to the CVODES 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`.

Some minor changes were made in order to minimize the differences between the sources for private functions in CVODES and CVODE.

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

In `CVodeQuadSensInit`, the line `cv_mem->cv_fQS_data = ...` was corrected (missing `Q`).

In the User Guide, a paragraph was added in Section 6.2.1 on `CVodeAdjReInit`, and a paragraph was added in Section 6.2.9 on `CVodeGetAdjY`. In the example `cvsRoberts_ASAi_dns`, the output was revised to include the use of `CVodeGetAdjY`.

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`.

For the Adjoint Sensitivity Analysis case in which the backward problem depends on the forward sensitivities, options have been added to allow for user-supplied `pset`, `psolve`, and `jtimes` functions.

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

In the example `cvsHessian_ASA_FSA`, an error was corrected in the function `fB2`: `y2` in place of `y3` in the third term of `Ith(yBdot,6)`.

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. In a minor change to the user interface, the type of the index `which` in CVODES was changed from `long int` to `int`.

Errors in the logic for the integration of backward problems were identified and fixed.

A large number of minor errors have been fixed. Among these are the following: 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 related to the integration of ODE IVP problems 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.

This version also includes several new features related to sensitivity analysis, among which are: (a) support for integration of quadrature equations depending on both the states and forward sensitivity (and thus support for forward sensitivity analysis of quadrature equations), (b) support for simultaneous integration of multiple backward problems based on the same underlying ODE (e.g., for use in an *forward-over-adjoint* method for computing second order derivative information), (c) support for backward integration of ODEs and quadratures depending on both forward states and sensitivities (e.g., for use in computing second-order derivative information), and (d) support for reinitialization of the adjoint module.

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. Moreover, the prototypes of all functions related to integration of backward problems were modified to support the simultaneous integration of multiple problems. All backward problems defined by the user are internally managed through a linked list and identified in the user interface through a unique identifier.

Changes in v2.5.0

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

In the adjoint solver module, the following two bugs were fixed: in `CVodeF` the solver was sometimes incorrectly taking an additional step before returning control to the user (in `CV_NORMAL` mode) thus leading to a failure in the interpolated output function; in `CVodeB`, while searching for the current check point, the solver was sometimes reaching outside the integration interval resulting in a segmentation fault.

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). 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.

A new interpolation method was added to the CVODES adjoint module. The function `CVadjMalloc` has an additional argument which can be used to select the desired interpolation scheme.

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 (`cvodes_` 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

A minor bug was fixed in the interpolation functions of the adjoint CVODES module.

Changes in v2.2.0

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 v2.1.2

A bug was fixed in the `CVode` function that was potentially leading to erroneous behaviour of the rootfinding procedure on the integration first step.

Changes in v2.1.1

This CVODES release includes bug fixes related to forward sensitivity computations (possible loss of accuray on a BDF order increase and incorrect logic in testing user-supplied absolute tolerances). In addition, we have added the option of activating and deactivating forward sensitivity calculations on successive CVODES runs without memory allocation/deallocation.

Other changes in this minor SUNDIALS release affect the build system.

Changes in v2.1.0

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, CVODES 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 §4.5.6 and §4.5.8.

Additionally, the interfaces to several user-supplied routines (such as those providing Jacobians, preconditioner information, and sensitivity right hand sides) 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 CVODES (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 CVODES. The most casual user, with a small IVP problem only, can get by with reading §2.1, then Chapter 4 through §4.5.5 only, and looking at examples in [35]. In addition, to solve a forward sensitivity problem the user should read §2.6, followed by Chapter 5 through §5.2.4 only, and look at examples in [35].

In a different direction, a more expert user with an IVP problem may want to (a) use a package preconditioner (§4.8), (b) supply his/her own Jacobian or preconditioner routines (§4.6), (c) do multiple runs of problems of the same size (§4.5.9), (d) supply a new NVECTOR module (Chapter 7), or even (e) supply new SUNLINSOL and/or SUNMATRIX modules (Chapters 8 and 9). An advanced user with a forward sensitivity problem may also want to (a) provide his/her own sensitivity equations right-hand side routine (§5.3), (b) perform multiple runs with the same number of sensitivity parameters (§5.2.1), or (c) extract additional diagnostic information (§5.2.4). A user with an adjoint sensitivity problem needs to understand the IVP solution approach at the desired level and also go through §2.7 for a short mathematical description of the adjoint approach, Chapter 6 for the usage of the adjoint module in CVODES, and the examples in [35].

The structure of this document is as follows:

- In Chapter 2, we give short descriptions of the numerical methods implemented by CVODES for the solution of initial value problems for systems of ODEs, continue with short descriptions of preconditioning (§2.2), stability limit detection (§2.3), and rootfinding (§2.4), and conclude with an overview of the mathematical aspects of sensitivity analysis, both forward (§2.6) and adjoint (§2.7).

- The following chapter describes the structure of the SUNDIALS suite of solvers (§3.1) and the software organization of the CVODES solver (§3.2).

- Chapter 4 is the main usage document for CVODES for simulation applications. It includes a complete description of the user interface for the integration of ODE initial value problems. Readers that are not interested in using CVODES for sensitivity analysis can then skip the next two chapters.

- Chapter 5 describes the usage of CVODES for forward sensitivity analysis as an extension of its IVP integration capabilities. We begin with a skeleton of the user main program, with emphasis on the steps that are required in addition to those already described in Chapter 4. Following that we provide detailed descriptions of the user-callable interface routines specific to forward sensitivity analysis and of the additonal optional user-defined routines.

- Chapter 6 describes the usage of CVODES for adjoint sensitivity analysis. We begin by describing the CVODES checkpointing implementation for interpolation of the original IVP solution during integration of the adjoint system backward in time, and with an overview of a user's main program. Following that we provide complete descriptions of the user-callable interface routines for adjoint sensitivity analysis as well as descriptions of the required additional user-defined routines.

- Chapter 7 gives a brief overview of the generic NVECTOR module shared among the various components of SUNDIALS, and details on the NVECTOR implementations provided with SUNDIALS.

- Chapter 8 gives a brief overview of the generic SUNMATRIX module shared among the various components of SUNDIALS, and details on the SUNMATRIX implementations provided with SUNDIALS: a dense implementation (§8.1), a banded implementation (§8.2) and a sparse implementation (§8.3).

- Chapter 9 gives a brief overview of the generic SUNLINSOL module shared among the various components of SUNDIALS. This chapter contains details on the SUNLINSOL implementations provided with SUNDIALS. The chapter also contains details on the SUNLINSOL implementations provided with SUNDIALS that interface with external linear solver libraries.

- Finally, in the appendices, we provide detailed instructions for the installation of CVODES, within the structure of SUNDIALS (Appendix A), as well as a list of all the constants used for input to and output from CVODES 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 CVDLS, are written in all capitals. Usage and installation instructions that constitute important warnings are marked with a triangular symbol ⚠ in the margin.

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

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

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

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. CVODES solves both stiff and non-stiff systems. Roughly speaking, stiffness is characterized by the presence of at least one rapidly damped mode, whose time constant is small compared to the time scale of the solution itself.

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

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

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

2.1 IVP solution

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

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

Here the y^n are computed approximations to $y(t_n)$, and $h_n = t_n - t_{n-1}$ is the step size. The user of CVODE must choose appropriately one of two multistep methods. For nonstiff problems, CVODE includes the Adams-Moulton formulas, characterized by $K_1 = 1$ and $K_2 = q$ above, where the order q varies between 1 and 12. For stiff problems, CVODES 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 [6] and [25].

For either choice of formula, the nonlinear system

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

where $a_n \equiv \sum_{i>0}(\alpha_{n,i} y^{n-i} + h_n \beta_{n,i} \dot{y}^{n-i})$, must be solved (approximately) at each integration step. For this, CVODES offers the choice of either *functional iteration*, suitable only for nonstiff systems, and various versions of *Newton iteration*. Functional iteration, given by

$$y^{n(m+1)} = h_n \beta_{n,0} f(t_n, y^{n(m)}) + a_n\,,$$

involves evaluations of f only. In contrast, Newton iteration requires the solution of linear systems

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

in which

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

The initial guess for the iteration is a predicted value $y^{n(0)}$ computed explicitly from the available history data.

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

- dense direct solvers, using either an internal implementation or a Blas/Lapack implementation (serial or threaded vector modules only),
- band direct solvers, using either an internal implementation or a Blas/Lapack implementation (serial or threaded vector modules only),
- sparse direct solver interfaces, using either the KLU sparse solver library [11, 1], or the thread-enabled SuperLU_MT sparse solver library [27, 12, 2] (serial or threaded vector modules only) [Note that users will need to download and install the KLU or SUPERLUMT packages independent of CVODES],
- 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 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 [4].

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, CVODES 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

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

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

In the cases of a direct solver (dense, band, sparse, or diagonal), the iteration is a Modified Newton iteration, in that the iteration matrix M is fixed throughout the nonlinear iterations. However, for any of the Krylov methods, it 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. The matrix M (direct cases) or preconditioner matrix P (Krylov cases) is 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 stopping test for the Newton iteration 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 (2.4), we want to ensure that the iteration error $y^n - y^{n(m)}$ is small relative to ϵ, specifically that it is less than 0.1ϵ. (The safety factor 0.1 can be changed by the user.) For this, we also estimate the linear convergence rate constant R as follows. We initialize R to 1, and reset $R = 1$ when M or P is updated. After computing a correction $\delta_m = y^{n(m)} - y^{n(m-1)}$, we update R if $m > 1$ as

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

Now we use the estimate

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

Therefore the convergence (stopping) test is

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

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

When a Krylov 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 $0.05 \cdot (0.1\epsilon)$.

With the direct dense and band methods, 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 the sparse direct solvers, 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

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

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

A critical part of CVODES — 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, CVODES returns to the user with a give-up message.

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

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

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

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

and

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

The new order and step size are then set according to

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

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

The various algorithmic features of CVODES described above, as inherited from VODE and VODPK, are documented in [3, 5, 19]. They are also summarized in [20].

CVODES 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, CVODES estimates a new step size h' using a linear approximation ofthe components in y that failed the constraint test (including a afety factor of 0.9 to cover the strict inequality case).

Normally, CVODES 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 CVODES not to integrate past a given stopping point $t = t_{\text{stop}}$.

2.2 Preconditioning

When using a Newton method to solve the nonlinear system (2.4), CVODES 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, 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_LP_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 [4] for an extensive study of preconditioners for reaction-transport systems).

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

Typical preconditioners used with CVODES are based on approximations to the system Jacobian, $J = \partial f/\partial y$. Since the Newton iteration 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 Newton 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

CVODES 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 [17]. 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 CVODES. 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 [18], where it works well. The implementation in CVODES 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 CVODES solver has been augmented to include a rootfinding feature. This means that, while integrating the Initial Value Problem (2.1), CVODES 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 CVODES. 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 [16]. In addition, each time g is computed, CVODES 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, CVODES 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, CVODES stops and reports an error. This way, each time CVODES 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, CVODES 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 $\tau/2$.

2.5 Pure quadrature integration

In many applications, and most notably during the backward integration phase of an adjoint sensitivity analysis run (see §2.7) it is of interest to compute integral quantities of the form

$$z(t) = \int_{t_0}^{t} q(\tau, y(\tau), p)\, d\tau\,. \tag{2.9}$$

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

$$\dot{z} = q(t, y, p)\,, \quad z(t_0) = 0\,. \tag{2.10}$$

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

This can be done at the "user level" by simply exposing to CVODES the extended ODE system (2.2)+(2.9). However, in the context of an implicit integration solver, this approach is not desirable since the nonlinear solver module will require the Jacobian (or Jacobian-vector product) of this extended ODE. Moreover, since the additional states z do not enter the right-hand side of the ODE (2.9) and therefore the right-hand side of the extended ODE system, it is much more efficient to treat the ODE system (2.9) separately from the original system (2.2) by "taking out" the additional states z from the nonlinear system (2.4) that must be solved in the correction step of the LMM. Instead, "corrected" values z^n are computed explicitly as

$$z^n = -\frac{1}{\alpha_{n,0}} \left(h_n \beta_{n,0} q(t_n, y_n, p) + h_n \sum_{i=1}^{K_2} \beta_{n,i} \dot{z}^{n-i} + \sum_{i=1}^{K_1} \alpha_{n,i} z^{n-i} \right)\,,$$

once the new approximation y^n is available.

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

2.6 Forward sensitivity analysis

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

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

$$\dot{s}_i = \frac{\partial f}{\partial y} s_i + \frac{\partial f}{\partial p_i}\,, \quad s_i(t_0) = \frac{\partial y_0(p)}{\partial p_i}\,, \tag{2.11}$$

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

When performing forward sensitivity analysis, CVODES carries out the time integration of the combined system, (2.2) and (2.11), by viewing it as an ODE system of size $N(N_s + 1)$, where N_s is the number of model parameters p_i, with respect to which sensitivities are desired ($N_s \leq N_p$). However, major improvements in efficiency can be made by taking advantage of the special form of the sensitivity equations as linearizations of the original ODEs. In particular, for stiff systems, for which CVODES employs a Newton iteration, the original ODE system and all sensitivity systems share the same Jacobian matrix, and therefore the same iteration matrix M in (2.6).

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

2.6.1 Forward sensitivity methods

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

- *Staggered Direct*

  In this approach [9], the nonlinear system (2.4) is first solved and, once an acceptable numerical solution is obtained, the sensitivity variables at the new step are found by directly solving (2.11) after the (BDF or Adams) discretization is used to eliminate $\dot{s}_i$. Although the system matrix of the above linear system is based on exactly the same information as the matrix M in (2.6), it must be updated and factored at every step of the integration, in contrast to an evalutaion of M which is updated only occasionally. For problems with many parameters (relative to the problem size), the staggered direct method can outperform the methods described below [26]. However, the computational cost associated with matrix updates and factorizations makes this method unattractive for problems with many more states than parameters (such as those arising from semidiscretization of PDEs) and is therefore not implemented in CVODES.

- *Simultaneous Corrector*

  In this method [28], the discretization is applied simultaneously to both the original equations (2.2) and the sensitivity systems (2.11) resulting in the following nonlinear system

  $$\hat{G}(\hat{y}_n) \equiv \hat{y}_n - h_n \beta_{n,0} \hat{f}(t_n, \hat{y}_n) - \hat{a}_n = 0\,,$$

  where $\hat{f} = [f(t,y,p), \ldots, (\partial f/\partial y)(t,y,p)s_i + (\partial f/\partial p_i)(t,y,p), \ldots]$, and $\hat{a}_n$ is comprised of the terms in the discretization that depend on the solution at previous integration steps. This combined nonlinear system can be solved using a modified Newton method as in (2.5) by solving the corrector equation

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

  at each iteration, where

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

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

- *Staggered corrector*

  In this approach [13], as in the staggered direct method, the nonlinear system (2.4) is solved first using the Newton iteration (2.5). Then a separate Newton iteration is used to solve the sensitivity system (2.11):

  $$M[s_i^{n(m+1)} - s_i^{n(m)}] = - \left[s_i^{n(m)} - \gamma \left(\frac{\partial f}{\partial y}(t_n, y^n, p) s_i^{n(m)} + \frac{\partial f}{\partial p_i}(t_n, y^n, p) \right) - a_{i,n} \right], \tag{2.13}$$

  where $a_{i,n} = \sum_{j>0}(\alpha_{n,j} s_i^{n-j} + h_n \beta_{n,j} \dot{s}_i^{n-j})$. In other words, a modified Newton iteration is used to solve a linear system. In this approach, the vectors $\partial f/\partial p_i$ need be updated only once per integration step, after the state correction phase (2.5) has converged. Note also that Jacobian-related data can be reused at all iterations (2.13) to evaluate the products $(\partial f/\partial y)s_i$.

CVODES implements the simultaneous corrector method and two flavors of the staggered corrector method which differ only if the sensitivity variables are included in the error control test. In the *full error control* case, the first variant of the staggered corrector method requires the convergence of the iterations (2.13) for all N_s sensitivity systems and then performs the error test on the sensitivity variables. The second variant of the method will perform the error test for each sensitivity vector $s_i, (i = 1, 2, \ldots, N_s)$ individually, as they pass the convergence test. Differences in performance between the two variants may therefore be noticed whenever one of the sensitivity vectors s_i fails a convergence or error test.

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

2.6.2 Selection of the absolute tolerances for sensitivity variables

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

2.6.3 Evaluation of the sensitivity right-hand side

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

Using central finite differences as an example, the two terms $(\partial f/\partial y)s_i$ and $\partial f/\partial p_i$ in the right-hand side of (2.11) can be evaluated either separately:

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

$$\frac{\partial f}{\partial p_i} \approx \frac{f(t, y, p + \sigma_i e_i) - f(t, y, p - \sigma_i e_i)}{2\,\sigma_i}\,, \tag{2.14'}$$

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

or simultaneously:

$$\frac{\partial f}{\partial y}s_i + \frac{\partial f}{\partial p_i} \approx \frac{f(t, y + \sigma s_i, p + \sigma e_i) - f(t, y - \sigma s_i, p - \sigma e_i)}{2\,\sigma}\,, \qquad (2.15)$$
$$\sigma = \min(\sigma_i, \sigma_y)\,,$$

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

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

$$\frac{\partial f}{\partial y}s_i + \frac{\partial f}{\partial p_i} \approx \frac{f(t, y + \sigma s_i, p + \sigma e_i) - f(t, y, p)}{\sigma}$$

can be used. In CVODES, the default value of $\rho_{\max} = 0$ indicates the use of the second-order centered directional derivative formula (2.15) exclusively. Otherwise, the magnitude of $\rho_{\max}$ and its sign (positive or negative) indicates whether this switching is done with regard to (centered or forward) finite differences, respectively.

2.6.4 Quadratures depending on forward sensitivities

If pure quadrature variables are also included in the problem definition (see §2.5), CVODES does *not* carry their sensitivities automatically. Instead, we provide a more general feature through which integrals depending on both the states y of (2.2) and the state sensitivities s_i of (2.11) can be evaluated. In other words, CVODES provides support for computing integrals of the form:

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

If the sensitivities of the quadrature variables z of (2.9) are desired, these can then be computed by using:

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

as integrands for $\bar{z}$, where q_y and q_p are the partial derivatives of the integrand function q of (2.9).

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

2.7 Adjoint sensitivity analysis

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

$$G(p) = \int_{t_0}^{T} g(t, y, p)dt\,, \qquad (2.16)$$

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

In what follows, we only sketch the analysis for the sensitivity problem for both G and g. For details on the derivation see [8]. Introducing a Lagrange multiplier λ, we form the augmented objective function

$$I(p) = G(p) - \int_{t_0}^{T} \lambda^* \left(\dot{y} - f(t, y, p)\right) dt \,, \tag{2.17}$$

where $*$ denotes the conjugate transpose. The gradient of G with respect to p is

$$\frac{dG}{dp} = \frac{dI}{dp} = \int_{t_0}^{T} (g_p + g_y s) dt - \int_{t_0}^{T} \lambda^* \left(\dot{s} - f_y s - f_p\right) dt \,, \tag{2.18}$$

where subscripts on functions f or g are used to denote partial derivatives and $s = [s_1, \ldots, s_{N_s}]$ is the matrix of solution sensitivities. Applying integration by parts to the term $\lambda^* \dot{s}$, and by requiring that λ satisfy

$$\begin{aligned} \dot{\lambda} &= -\left(\frac{\partial f}{\partial y}\right)^* \lambda - \left(\frac{\partial g}{\partial y}\right)^* \\ \lambda(T) &= 0 \,, \end{aligned} \tag{2.19}$$

the gradient of G with respect to p is nothing but

$$\frac{dG}{dp} = \lambda^*(t_0) s(t_0) + \int_{t_0}^{T} \left(g_p + \lambda^* f_p\right) dt \,. \tag{2.20}$$

The gradient of $g(T, y, p)$ with respect to p can be then obtained by using the Leibnitz differentiation rule. Indeed, from (2.16),

$$\frac{dg}{dp}(T) = \frac{d}{dT}\frac{dG}{dp}$$

and therefore, taking into account that dG/dp in (2.20) depends on T both through the upper integration limit and through λ, and that $\lambda(T) = 0$,

$$\frac{dg}{dp}(T) = \mu^*(t_0) s(t_0) + g_p(T) + \int_{t_0}^{T} \mu^* f_p dt \,, \tag{2.21}$$

where μ is the sensitivity of λ with respect to the final integration limit T. Thus μ satisfies the following equation, obtained by taking the total derivative with respect to T of (2.19):

$$\begin{aligned} \dot{\mu} &= -\left(\frac{\partial f}{\partial y}\right)^* \mu \\ \mu(T) &= \left(\frac{\partial g}{\partial y}\right)^*_{t=T} \,. \end{aligned} \tag{2.22}$$

The final condition on $\mu(T)$ follows from $(\partial\lambda/\partial t) + (\partial\lambda/\partial T) = 0$ at T, and therefore, $\mu(T) = -\dot{\lambda}(T)$.

The first thing to notice about the adjoint system (2.19) is that there is no explicit specification of the parameters p; this implies that, once the solution λ is found, the formula (2.20) can then be used to find the gradient of G with respect to any of the parameters p. The same holds true for the system (2.22) and the formula (2.21) for gradients of $g(T, y, p)$. The second important remark is that the adjoint systems (2.19) and (2.22) are terminal value problems which depend on the solution $y(t)$ of the original IVP (2.2). Therefore, a procedure is needed for providing the states y obtained during a forward integration phase of (2.2) to CVODES during the backward integration phase of (2.19) or (2.22). The approach adopted in CVODES, based on *checkpointing*, is described below.

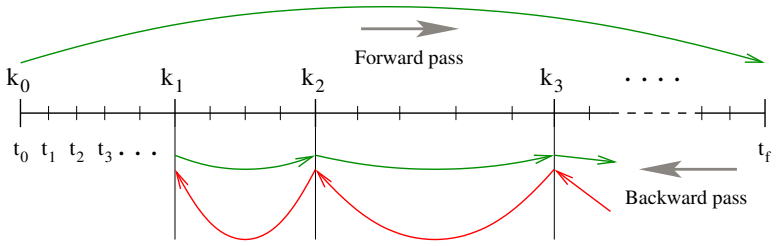


Figure 2.1: Illustration of the checkpointing algorithm for generation of the forward solution during the integration of the adjoint system.

2.7.1 Checkpointing scheme

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

However, especially for large-scale problems and long integration intervals, the number and size of the vectors y and $\dot{y}$ that would need to be stored make this approach computationally intractable. Thus, CVODES settles for a compromise between storage space and execution time by implementing a so-called *checkpointing scheme*. At the cost of at most one additional forward integration, this approach offers the best possible estimate of memory requirements for adjoint sensitivity analysis. To begin with, based on the problem size N and the available memory, the user decides on the number N_d of data pairs $(y, \dot{y})$ if cubic Hermite interpolation is selected, or on the number N_d of y vectors in the case of variable-degree polynomial interpolation, that can be kept in memory for the purpose of interpolation. Then, during the first forward integration stage, after every N_d integration steps a checkpoint is formed by saving enough information (either in memory or on disk) to allow for a hot restart, that is a restart which will exactly reproduce the forward integration. In order to avoid storing Jacobian-related data at each checkpoint, a reevaluation of the iteration matrix is forced before each checkpoint. At the end of this stage, we are left with N_c checkpoints, including one at t_0. During the backward integration stage, the adjoint variables are integrated from T to t_0 going from one checkpoint to the previous one. The backward integration from checkpoint $i+1$ to checkpoint i is preceded by a forward integration from i to $i+1$ during which the N_d vectors y (and, if necessary $\dot{y}$) are generated and stored in memory for interpolation[1] (see Fig. 2.1).

This approach transfers the uncertainty in the number of integration steps in the forward integration phase to uncertainty in the final number of checkpoints. However, N_c is much smaller than the number of steps taken during the forward integration, and there is no major penalty for writing/reading the checkpoint data to/from a temporary file. Note that, at the end of the first forward integration stage, interpolation data are available from the last checkpoint to the end of the interval of integration. If no checkpoints are necessary (N_d is larger than the number of integration steps taken in the solution of (2.2)), the total cost of an adjoint sensitivity computation can be as low as one

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

The Hermite cubic interpolation option is present because it was implemented chronologically first and it is also used by other adjoint solvers (e.g. DASPKADJOINT). The variable-degree polynomial is more memory-efficient (it requires only half of the memory storage of the cubic Hermite interpolation) and is more accurate. The accuracy differences are minor when using BDF (since the maximum method order cannot exceed 5), but can be significant for the Adams method for which the order can reach 12.

forward plus one backward integration. In addition, CVODES provides the capability of reusing a set of checkpoints for multiple backward integrations, thus allowing for efficient computation of gradients of several functionals (2.16).

Finally, we note that the adjoint sensitivity module in CVODES provides the necessary infrastructure to integrate backwards in time any ODE terminal value problem dependent on the solution of the IVP (2.2), including adjoint systems (2.19) or (2.22), as well as any other quadrature ODEs that may be needed in evaluating the integrals in (2.20) or (2.21). In particular, for ODE systems arising from semi-discretization of time-dependent PDEs, this feature allows for integration of either the discretized adjoint PDE system or the adjoint of the discretized PDE.

2.8 Second-order sensitivity analysis

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

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

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

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

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

A much more efficient alternative is to compute Hessian-vector products using a so-called *forward-over-adjoint* approach. This method is based on using the same "trick" as the one used in computing gradients of pointwise functionals with the adjoint method, namely applying a formal directional forward derivation to one of the gradients of (2.20) or (2.21). With that, the cost of computing a full Hessian is roughly equivalent to the cost of computing the gradient with forward sensitivity analysis. However, Hessian-vector products can be cheaply computed with one additional adjoint solve. Consider for example, $G(p) = \int_{t_0}^{t_f} g(t, y)\, dt$. It can be shown that the product between the Hessian of G (with respect to the parameters p) and some vector u can be computed as

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

where λ, μ, and s are solutions of

$$\begin{aligned} -\dot{\mu} &= f_y^T \mu + \left(\lambda^T \otimes I_n\right) f_{yy} s + g_{yy} s \,; \quad \mu(t_f) = 0 \\ -\dot{\lambda} &= f_y^T \lambda + g_y^T \,; \quad \lambda(t_f) = 0 \\ \dot{s} &= f_y s \,; \quad s(t_0) = y_{0p} u \end{aligned} \tag{2.23}$$

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

[2] For the sake of simplifity in presentation, we do not include explicit dependencies of g on time t or parameters p. Moreover, we only consider the case in which the dependency of the original ODE (2.2) on the parameters p is through its initial conditions only. For details on the derivation in the general case, see [29].

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

Therefore, the cost of computing the Hessian-vector product is roughly that of two forward and two backward integrations of a system of ODEs of size N. For more details, including the corresponding formulas for a pointwise model functional output, see [29].

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

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

Chapter 3

Code Organization

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 $M dy/dt = f_E(t, y) + f_I(t, y)$ 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 CVODES organization

The CVODES 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 CVODES package is shown in Figure 3.3. The basic elements of the structure are a module for the basic integration algorithm (including forward sensitivity analysis), a module for adjoint sensitivity analysis, and support for the solution of linear systems that arise in the case of a stiff system. The central integration module, implemented in the files `cvode.h`, `cvode_impl.h`, and `cvode.c`, deals with the evaluation of integration coefficients, the functional or Newton iteration process, estimation of local error, selection of stepsize and order, and interpolation to user output points, among other issues. Although this module contains logic for the basic Newton iteration algorithm, it has no knowledge of the method being used to solve the linear systems that arise. For any given user problem, one of the linear system solver interfaces is specified, and is then invoked as needed during the integration.

In addition, if forward sensitivity analysis is turned on, the main module will integrate the forward sensitivity equations simultaneously with the original IVP. The sensitivity variables may be included in the local error control mechanism of the main integrator. CVODES provides three different strategies

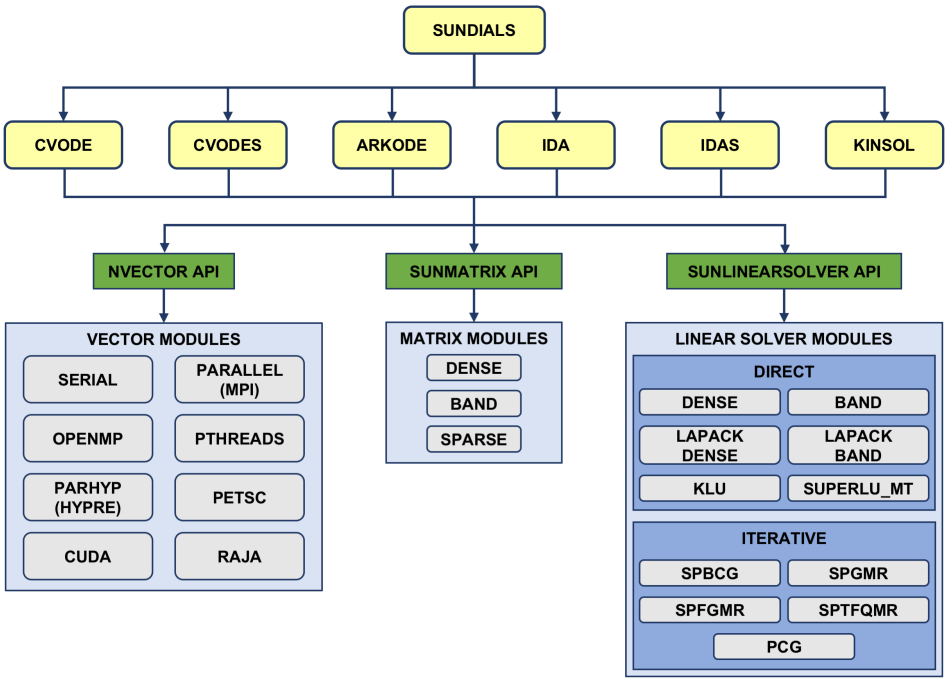


Figure 3.1: High-level diagram of the SUNDIALS suite

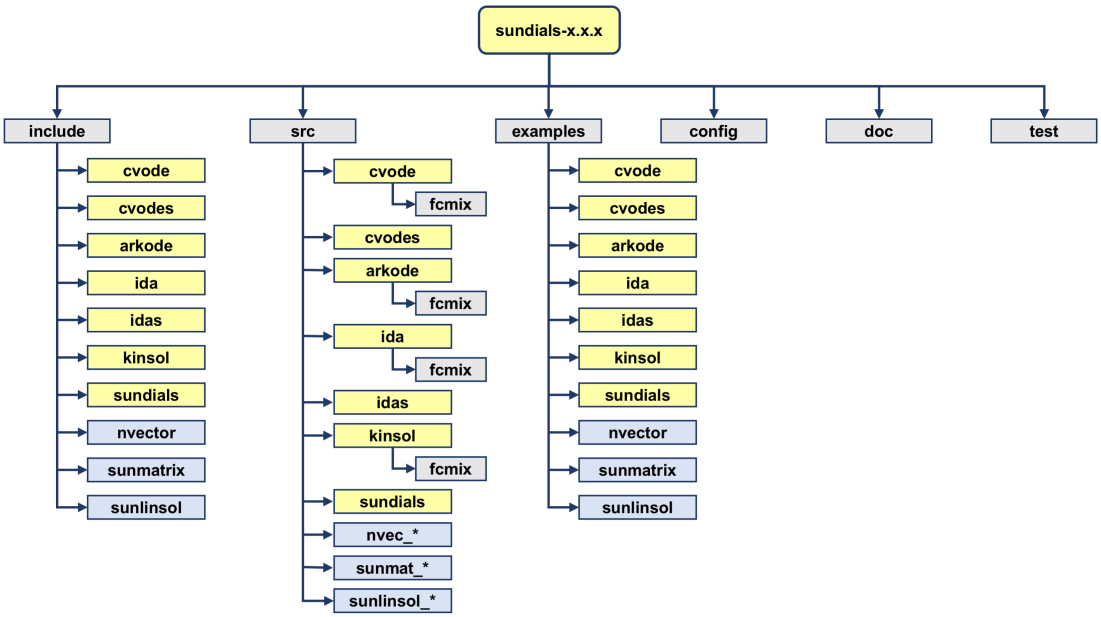
for dealing with the correction stage for the sensitivity variables: `CV_SIMULTANEOUS`, `CV_STAGGERED` and `CV_STAGGERED1` (see §2.6 and §5.2.1). The CVODES package includes an algorithm for the approximation of the sensitivity equations right-hand sides by difference quotients, but the user has the option of supplying these right-hand sides directly.

The adjoint sensitivity module (file `cvodea.c`) provides the infrastructure needed for the backward integration of any system of ODEs which depends on the solution of the original IVP, in particular the adjoint system and any quadratures required in evaluating the gradient of the objective functional. This module deals with the setup of the checkpoints, the interpolation of the forward solution during the backward integration, and the backward integration of the adjoint equations.

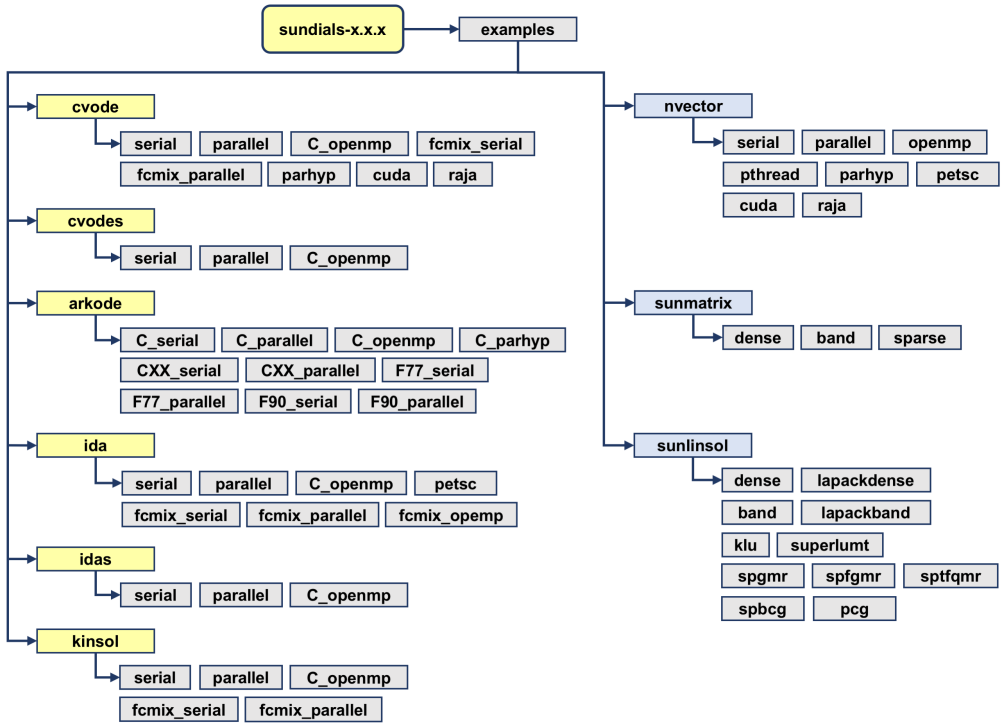
At present, the package includes two linear solver interfaces. The *direct* linear solver interface, CVDLS, supports SUNLINSOL implementations with type `SUNLINSOL_DIRECT` (see Chapter 9). These linear solvers utilize direct methods for the solution of linear systems stored using one of the SUNDIALS generic SUNMATRIX implementations (dense, banded or sparse; see Chapter 8). It is assumed that the dominant cost for such solvers occurs in factorization of the linear system matrix M, so CVODE utilizes these solvers within its modified Newton nonlinear solve. The *spils* linear solver interface, CVSPILS, supports SUNLINSOL implementations with type `SUNLINSOL_ITERATIVE` (see Chapter 9). These linear solvers utilize scaled preconditioned iterative methods. It is assumed that these methods are implemented in a "matrix-free" manner, wherein only the action of the matrix-vector product Mv is required. Since CVODE can operate on any valid SUNLINSOL implementation of `SUNLINSOL_DIRECT` or `SUNLINSOL_ITERATIVE` types, the set of linear solver modules available to CVODES will expand as new SUNLINSOL modules are developed.

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

Within the CVDLS interface, the package includes algorithms for the approximation of dense or banded Jacobians through difference quotients, but the user also has the option of supplying the Jacobian (or an approximation to it) directly. This user-supplied routine is required when using sparse Jacobian matrices, since standard difference quotient approximations do not leverage the inherent



(a) Directory structure of the SUNDIALS source tree



(b) Directory structure of the SUNDIALS examples

Figure 3.2: Organization of the SUNDIALS suite

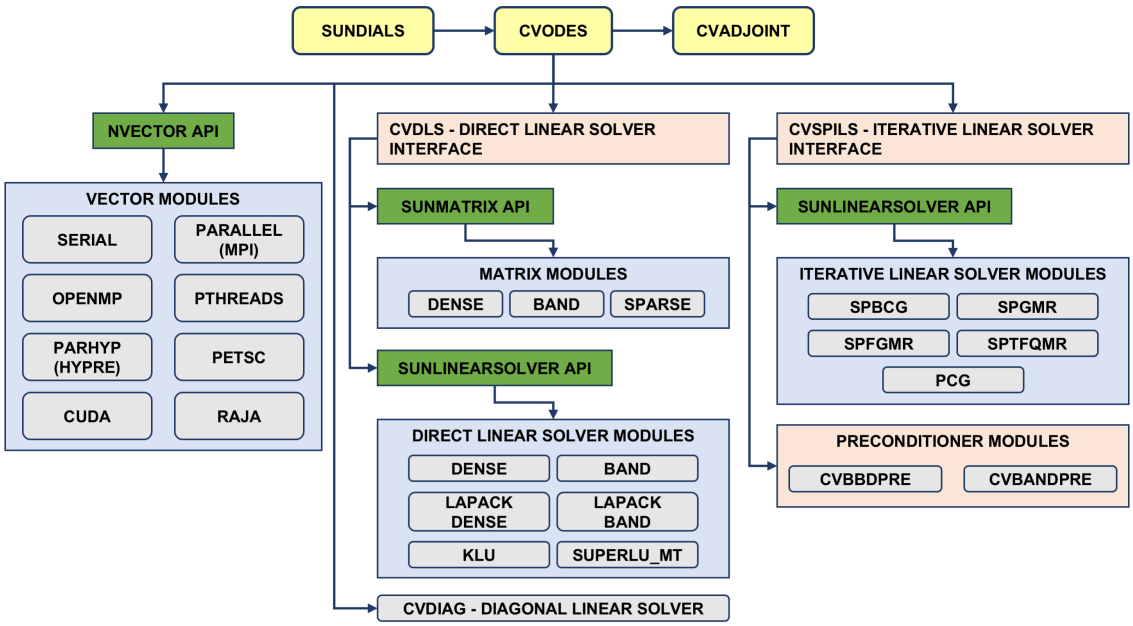


Figure 3.3: Overall structure of the CVODES package. Modules specific to CVODES begin with "CV" (CVDLS, CVDIAG, CVSPILS, CVBBDPRE and CVBANDPRE), all other items correspond to generic solver and auxiliary modules. Note also that the LAPACK, KLU and SUPERLUMT support is through interfaces to external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

sparsity of the problem.

Within the CVSPILS interface, the package 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 [4, 5], together with the example and demonstration programs included with CVODES, offer considerable assistance in building preconditioners.

Each CVODE 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.

CVODES also provides two preconditioner modules, for use with any of the Krylov iterative linear solvers. The first one, CVBANDPRE, is intended to be used with NVECTOR_SERIAL, NVECTOR_OPENMP or NVECTOR_PTHREADS and provides a banded difference-quotient Jacobian-based preconditioner, with corresponding setup and solve routines. The second preconditioner module, CVBBDPRE, works in conjunction with NVECTOR_PARALLEL and generates a preconditioner that is a block-diagonal matrix with each block being a banded matrix.

All state information used by CVODES 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 CVODES 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 CVODES memory structure. The reentrancy of CVODES 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 CVODES for IVP Solution

This chapter is concerned with the use of CVODES for the solution of initial value problems (IVPs). The following sections treat the header files and the layout of the user's main program, and provide descriptions of the CVODES user-callable functions and user-supplied functions. This usage is essentially equivalent to using CVODE [21].

The sample programs described in the companion document [35] 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 CVODES package.

The user should be aware that not all SUNLINSOL and SUNMATRIX modules are compatible with all NVECTOR implementations. Details on compatability are given in the documentation for each SUNMATRIX module (Chapter 8) and each SUNLINSOL module (Chapter 9). For example, NVECTOR_PARALLEL is not compatible with the dense, banded, or sparse SUNMATRIX types, or with the corresponding dense, banded, or sparse SUNLINSOL modules. Please check Chapters 8 and 9 to verify compatability 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.

CVODES 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 CVODES, 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 CVODES. The relevant library files are

- *libdir*/`libsundials_cvodes.`*lib*,
- *libdir*/`libsundials_nvec*.`*lib* (one to four files),

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/cvodes`
- *incdir*/`include/sundials`
- *incdir*/`include/nvector`

- *incdir*/include/sunmatrix
- *incdir*/include/sunlinsol

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

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

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:

- `cvodes.h`, the main header file for CVODES, which defines the several types and various constants, and includes function prototypes.

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

The calling program must also include an NVECTOR implementation header file, of the form `nvector/nvector_***.h`. See Chapter 7 for the appropriate name. This file in turn includes the header file `sundials_nvector.h` which defines the abstract `N_Vector` data type.

If the user chooses Newton iteration for the solution of the nonlinear systems, then a linear solver module header file will be required. The header files corresponding to the various linear solver interfaces and linear solver modules available for use with CVODES are:

- `cvodes/cvodes_direct.h`, which is used with the CVDLS direct linear solver interface to access direct solvers with the following header files:
  - `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_superlumt.h`, which is used with the SUPERLUMT sparse linear solver interface module, SUNLINSOL_SUPERLUMT;
- `cvodes/cvodes_spils.h`, which is used with the CVSPILS iterative linear solver interface to access iterative solvers with the following header files:
  - `sunlinsol/sunlinsol_spgmr.h`, which is used with the scaled, preconditioned GMRES Krylov linear solver module, SUNLINSOL_SPGMR;
  - `sunlinsol/sunlinsol_spfgmr.h`, which is used with the scaled, preconditioned FGMRES Krylov linear solver module, SUNLINSOL_SPFGMR;
  - `sunlinsol/sunlinsol_spbcgs.h`, which is used with the scaled, preconditioned Bi-CGStab Krylov linear solver module, SUNLINSOL_SPBCGS;
  - `sunlinsol/sunlinsol_sptfqmr.h`, which is used with the scaled, preconditioned TFQMR Krylov linear solver module, SUNLINSOL_SPTFQMR;

  - sunlinsol/sunlinsol_pcg.h, which is used with the scaled, preconditioned CG Krylov linear solver module, SUNLINSOL_PCG;

- cvodes/cvodes_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 as well as various functions and macros acting on such matrices.

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

The header files for the SUNLINSOL_KLU and SUNLINSOL_SUPERLUMT sparse linear solvers include the file sunmatrix/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 cvsDiurnal_kry_p example (see [35]), 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, and SUNLINSOL implementations used. For the steps that are not, refer to Chapters 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 §7.1-7.4 for details.

   For the *hypre* and PETSc vector wrappers, first create and initialize the underlying vector, and then create an NVECTOR wrapper with a call of the form y0 = N_VMake_***(yvec), where yvec is a *hypre* or PETSc vector. Note that calls like N_VNew_***(...) and N_VGetArrayPointer(...) are not available for these vector wrappers. See §7.5 and §7.6 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 §7.7-7.8 for details.

4. **Create CVODES object**

   Call `cvode_mem = CVodeCreate(lmm, iter)` to create the CVODES memory block and to specify the solution method (linear multistep method and nonlinear solver iteration type). `CVodeCreate` returns a pointer to the CVODES memory structure. See §4.5.1 for details.

5. **Initialize CVODES solver**

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

6. **Specify integration tolerances**

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

7. **Set optional inputs**

   Call `CVodeSet*` functions to change any optional inputs that control the behavior of CVODES from their default values. See §4.5.6.1 for details.

8. **Create matrix object**

   If a direct linear solver is to be used within a Newton iteration then a template Jacobian matrix must be created by using the appropriate functions defined by the particular SUNMATRIX implementation.

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

9. **Create linear solver object**

   If a Newton iteration is chosen, then the desired linear solver object must be created by using the appropriate functions defined by the particular SUNLINSOL implementation.

10. **Set linear solver optional inputs**

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

11. **Attach linear solver module**

    If a Newton iteration is chosen, initialize the CVDLS or CVSPILS linear solver interface by attaching the linear solver object (and matrix object, if applicable) with one of the following calls (for details see §4.5.3):

    ```
    ier = CVDlsSetLinearSolver(...);
    ```

    ```
    ier = CVSpilsSetLinearSolver(...);
    ```

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

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

12. **Set linear solver interface optional inputs**

Call `CVDlsSet*` or `CVSpilsSet*` functions to change optional inputs specific to that linear solver interface. See §4.5.6 for details.

13. **Specify rootfinding problem**

    Optionally, call `CVodeRootInit` to initialize a rootfinding problem to be solved during the integration of the ODE system. See §4.5.4, and see §4.5.6.4 for relevant optional input calls.

14. **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 §4.5.5 for details.

15. **Get optional outputs**

    Call `CV*Get*` functions to obtain optional output. See §4.5.8 for details.

16. **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);
    ```

17. **Free solver memory**

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

18. **Free linear solver and matrix memory**

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

19. **Finalize MPI, if used**

    Call `MPI_Finalize()` to terminate MPI.

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

4.5 User-callable functions

This section describes the CVODES functions that are called by the user to setup and then solve an IVP. Some of these are required. However, starting with §4.5.6, the functions listed involve optional inputs/outputs or restarting, and those paragraphs may be skipped for a casual use of CVODES. 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 §4.5.6.1).

4.5.1 CVODES 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 CVODES memory block created and allocated by the first

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

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

two calls.

`CVodeCreate`

Call `cvode_mem = CVodeCreate(lmm, iter);`

Description The function `CVodeCreate` instantiates a CVODES solver object and specifies the solution method.

Arguments
- `lmm` (`int`) specifies the linear multistep method and may be one of two possible values: `CV_ADAMS` or `CV_BDF`.
- `iter` (`int`) specifies the type of nonlinear solver iteration and may be either `CV_NEWTON` or `CV_FUNCTIONAL`.

The recommended choices for (`lmm`, `iter`) are (`CV_ADAMS`, `CV_FUNCTIONAL`) for nonstiff problems and (`CV_BDF`, `CV_NEWTON`) for stiff problems.

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

`CVodeInit`

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

Description The function `CVodeInit` provides required problem and solution specifications, allocates internal memory, and initializes CVODES.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block returned by `CVodeCreate`.
- `f` (`CVRhsFn`) is the C function which computes the right-hand side function f in the ODE. This function has the form `f(t, y, ydot, user_data)` (for full details see §4.6.1).
- `t0` (`realtype`) is the initial value of t.
- `y0` (`N_Vector`) is the initial value of y.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeInit` was successful.
- `CV_MEM_NULL` The CVODES memory block was not initialized through a previous call to `CVodeCreate`.

`CV_MEM_FAIL`	A memory allocation request has failed.
`CV_ILL_INPUT`	An input argument to `CVodeInit` has an illegal value.

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

`CVodeFree`

Call `CVodeFree(&cvode_mem);`

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

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

Return value The function `CVodeFree` has no return value.

4.5.2 CVODES 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 CVODES 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 CVODES 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 CVODES 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 CVODES 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 CVODES 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 CVODES 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 1.0E-15).

(2) The absolute tolerances `abstol` (whether scalar or vector) need to be set to control absolute errors when any components of the solution vector `y` may be so small that pure relative error control is meaningless. For example, if `y[i]` starts at some nonzero value, but in time decays to zero, then pure relative error control on `y[i]` makes no sense (and is overly costly) after `y[i]` is below some noise level. Then `abstol` (if scalar) or `abstol[i]` (if a vector) needs to be set to that noise level. If the different components have different noise levels, then `abstol` should be a vector. See the example `cvsRoberts_dns` in the CVODES package, and the discussion of it in the CVODES Examples document [35]. In that problem, the three components vary betwen 0 and 1, and have different noise levels; hence the `abstol` vector. It is impossible to give any general advice on `abstol` values, because the appropriate noise levels are completely problem-dependent. The user or modeler hopefully has some idea as to what those noise levels are.

(3) Finally, it is important to pick all the tolerance values conservatively, because they control the error committed on each individual time step. The final (global) errors are some sort of accumulation of those per-step errors. A good rule of thumb is to reduce the tolerances by a factor of .01 from the actual desired limits on errors. So if you want .01% accuracy (globally), a good choice is `reltol` $= 10^{-6}$. But in any case, it is a good idea to do a few experiments with the tolerances to see how the computed solution values vary as tolerances are reduced.

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

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

(2) If output plots or tables are being generated, and it is important to avoid having negative numbers appear there (for the sake of avoiding a long explanation of them, if nothing else), then eliminate them, but only in the context of the output medium. Then the internal values carried by the solver are unaffected. Remember that a small negative value in `y` returned by CVODES, 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, a Newton iteration requires the solution of linear systems of the form (2.5). There are three CVODES linear solver interfaces currently available for this task: CVDLS, CVDIAG and CVSPILS.

The first corresponds to the use of Direct Linear Solvers, and utilizes SUNMATRIX objects to store the Jacobian $J = \partial f/\partial y$, the Newton matrix $M = I - \gamma J$, and factorizations used throughout the solution process.

The CVDIAG linear solver is also a direct linear solver, but it only uses a diagonal approximation to J.

The third corresponds to the use of Scaled, Preconditioned, Iterative Linear Solvers, utilizing matrix-free Krylov methods to solve the Newton linear systems of equations. With most of these methods, preconditioning can be done on the left only, on the right only, on both the left and the right, or not at all. The exceptions to this rule are SPFGMR that supports right preconditioning only and PCG that performs symmetric preconditioning. For the specification of a preconditioner, see the iterative linear solver sections in §4.5.6 and §4.6.

If preconditioning is done, user-supplied functions define 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 Newton matrix $M = I - \gamma J$ of (2.6).

To specify a generic linear solver to CVODES, after the call to `CVodeCreate` but before any calls to `CVodes`, the user's program must create the appropriate SUNLINSOL object and call either of the functions `CVDlsSetLinearSolver` or `CVSpilsSetLinearSolver`, as documented below. The first argument passed to these functions is the CVODES memory pointer returned by `CVodeCreate`; the second argument passed to these functions is the desired SUNLINSOL object to use for solving Newton systems. A call to one of these functions initializes the appropriate CVODES linear solver interface, linking this to the main CVODES integrator, and allows the user to specify parameters which are specific to a particular solver interface. The use of each of the generic linear solvers involves certain constants and possibly some macros, that are likely to be needed in the user code. These are available in the corresponding header file associated with the specific SUNMATRIX or SUNLINSOL module in question, as described in Chapters 8 and 9.

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

`CVDlsSetLinearSolver`

Call `flag = CVDlsSetLinearSolver(cvode_mem, LS, J);`

Description The function `CVDlsSetLinearSolver` attaches a direct SUNLINSOL object `LS` and corresponding template Jacobian SUNMATRIX object `J` to CVODES, initializing the CVDLS direct linear solver interface.

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `LS` (`SUNLinearSolver`) SUNLINSOL object to use for solving Newton linear systems.
- `J` (`SUNMatrix`) SUNMATRIX object for used as a template for the Jacobian (must have a type compatible with the linear solver object).

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

`CVDLS_SUCCESS` The CVDLS initialization was successful.

`CVDLS_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CVDLS_ILL_INPUT` The CVDLS solver is not compatible with the `LS` or `J` input objects or is incompatible with the current NVECTOR module.

`CVDLS_MEM_FAIL` A memory allocation request failed.

Notes The CVDLS linear solver interface is not compatible with all implementations of the SUNLINSOL and NVECTOR modules. Specifically, CVDLS requires use of a *direct* SUNLINSOL object and a serial or theaded NVECTOR module. Additional compatibility limitations for each SUNLINSOL object (i.e. SUNMATRIX and NVECTOR object compatibility) are described in Chapter 9.

`CVSpilsSetLinearSolver`

Call `flag = CVSpilsSetLinearSolver(cvode_mem, LS);`

Description The function `CVSpilsSetLinearSolver` attaches an iterative SUNLINSOL object `LS` to CVODES, initializing the CVSPILS scaled, preconditioned, iterative linear solver interface.

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

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

`LS` (`SUNLinearSolver`) SUNLINSOL object to use for solving Newton linear systems.

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

`CVSPILS_SUCCESS` The CVSPILS initialization was successful.

`CVSPILS_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CVSPILS_ILL_INPUT` The CVSPILS solver is not compatible with the `LS` object or is incompatible with the current NVECTOR module.

`CVSPILS_MEM_FAIL` A memory allocation request failed.

`CVSPILS_SUNLS_FAIL` A call to the `LS` object failed.

Notes The CVSPILS linear solver interface is not compatible with all implementations of the SUNLINSOL and NVECTOR modules. Specifically, CVSPILS requires use of an *iterative* SUNLINSOL object. Additional compatibility limitations for each SUNLINSOL object (i.e. required NVECTOR routines) are described in Chapter 9.

`CVDiag`

Call `flag = CVDiag(cvode_mem);`

Description The function `CVDiag` selects the CVDIAG linear solver.

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

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

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

`CVDIAG_SUCCESS` The CVDIAG initialization was successful.

`CVDIAG_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CVDIAG_ILL_INPUT` The CVDIAG solver is not compatible with the current NVECTOR module.

`CVDIAG_MEM_FAIL` A memory allocation request failed.

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

4.5.4 Rootfinding initialization function

While solving the IVP, CVODES 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 CVODES 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`$= 0$.

4.5.5 CVODES 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 CVODES is to return a solution. But these modes are modified if the user has set a stop time (with `CVodeSetStopTime`) or requested rootfinding.

`CVode`

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

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `tout` (`realtype`) the next time at which a computed solution is desired.
- `yout` (`N_Vector`) the computed solution vector.
- `tret` (`realtype`) the time reached by the solver (output).
- `itask` (`int`) a flag indicating the job of the solver for the next user step. The `CV_NORMAL` option causes the solver to take internal steps until it has reached or just passed the user-specified `tout` parameter. The solver then interpolates in order to return an approximate value of y(`tout`). The `CV_ONE_STEP` option tells the solver to take just one internal step and then return the solution at the point reached by that step.

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

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

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

`CV_SUCCESS` `CVode` succeeded and no roots were found.

CV_TSTOP_RETURN	CVode succeeded by reaching the stopping point specified through the optional input function CVodeSetStopTime (see §4.5.6.1).
CV_ROOT_RETURN	CVode succeeded and found one or more roots. In this case, tret is the location of the root. If nrtfn > 1, call CVodeGetRootInfo to see which g_i were found to have a root.
CV_MEM_NULL	The cvode_mem argument was NULL.
CV_NO_MALLOC	The CVODES memory was not allocated by a call to CVodeInit.
CV_ILL_INPUT	One of the inputs to CVode was illegal, or some other input to the solver was either illegal or missing. The latter category includes the following situations: (a) The tolerances have not been set. (b) A component of the error weight vector became zero during internal time-stepping. (c) The linear solver initialization function (called by the user after calling CVodeCreate) failed to set the linear solver-specific lsolve field in cvode_mem. (d) A root of one of the root functions was found both at a point t and also very near t. In any case, the user should see the error message for details.
CV_TOO_CLOSE	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.
CV_TOO_MUCH_WORK	The solver took mxstep internal steps but still could not reach tout. The default value for mxstep is MXSTEP_DEFAULT = 500.
CV_TOO_MUCH_ACC	The solver could not satisfy the accuracy demanded by the user for some internal step.
CV_ERR_FAILURE	Either error test failures occurred too many times (MXNEF = 7) during one internal time step, or with $\|h\| = h_{min}$.
CV_CONV_FAILURE	Either convergence test failures occurred too many times (MXNCF = 10) during one internal time step, or with $\|h\| = h_{min}$.
CV_LINIT_FAIL	The linear solver's initialization function failed.
CV_LSETUP_FAIL	The linear solver's setup function failed in an unrecoverable manner.
CV_LSOLVE_FAIL	The linear solver's solve function failed in an unrecoverable manner.
CV_CONSTR_FAIL	The inequality constraints were violated and the solver was unable to recover.
CV_RHSFUNC_FAIL	The right-hand side function failed in an unrecoverable manner.
CV_FIRST_RHSFUNC_FAIL	The right-hand side function had a recoverable error at the first call.
CV_REPTD_RHSFUNC_ERR	Convergence test failures occurred too many times due to repeated recoverable errors in the right-hand side function. This flag will also be returned if the right-hand side function had repeated recoverable errors during the estimation of an initial step size.
CV_UNREC_RHSFUNC_ERR	The right-hand function had a recoverable error, but no recovery was possible. This failure mode is rare, as it can occur only if the right-hand side function fails recoverably after an error test failed while at order one.
CV_RTFUNC_FAIL	The rootfinding function failed.

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.

Table 4.2: Optional inputs for CVODES, CVDLS, and CVSPILS

Optional input	Function name	Default
CVODES 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
Nonlinear iteration type	`CVodeSetIterType`	none
Inequality constraints on solution	`CVodeSetConstraints`	`NULL`
Direction of zero-crossing	`CVodeSetRootDirection`	both
Disable rootfinding warnings	`CVodeSetNoInactiveRootWarn`	none
CVDLS linear solver interface		
Jacobian function	`CVDlsSetJacFn`	DQ
CVSPILS linear solver interface		
Preconditioner functions	`CVSpilsSetPreconditioner`	NULL, NULL
Jacobian-times-vector functions	`CVSpilsSetJacTimes`	NULL, DQ
Ratio between linear and nonlinear tolerances	`CVSpilsSetEpsLin`	0.05

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.

4.5.6 Optional input functions

There are numerous optional input parameters that control the behavior of the CVODES solver. CVODES 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 CVODES which are then described in detail in the remainder of this section, begining with those for the main CVODES 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 CVODES, 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.

4.5.6.1 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 CVODES messages should be directed when the default CVODES error handler function is used.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`errfp` (`FILE *`) pointer to output file.

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

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

Notes The default value for `errfp` is `stderr`.

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

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

`CVodeSetErrHandlerFn`

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

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

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`ehfun` (`CVErrHandlerFn`) is the C error handler function (see §4.6.2).
`eh_data` (`void *`) pointer to user data passed to `ehfun` every time it is called.

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

`CV_SUCCESS` The function `ehfun` and data pointer `eh_data` have been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.

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

`CVodeSetUserData`

Call `flag = CVodeSetUserData(cvode_mem, user_data);`

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

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`user_data` (`void *`) pointer to the user data.

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

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

Notes If specified, the pointer to `user_data` is passed to all user-supplied functions that have it as an argument. Otherwise, a `NULL` pointer is passed.

If `user_data` is needed in user linear solver or preconditioner functions, the call to `CVodeSetUserData` must be made *before* the call to specify the linear solver.

`CVodeSetMaxOrd`

Call `flag = CVodeSetMaxOrd(cvode_mem, maxord);`

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

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`maxord` (`int`) value of the maximum method order. This must be positive.

Return value The return value `flag` (of type `int`) is one of
`CV_SUCCESS` The optional value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CV_ILL_INPUT` The specified value `maxord` is ≤ 0, or larger than its previous value.

Notes The default value is `ADAMS_Q_MAX = 12` for the Adams-Moulton method and `BDF_Q_MAX = 5` for the BDF method. Since `maxord` affects the memory requirements for the internal CVODES 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 CVODES 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 CVODES 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 CVODES 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 CVODES 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 `flag = CVodeSetInitStep(cvode_mem, hin);`

Description The function `CVodeSetInitStep` specifies the initial step size.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`hin` (`realtype`) value of the initial step size to be attempted. Pass 0.0 to use the default value.

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

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

Notes By default, CVODES estimates the initial step size to be the solution h of the equation $\|0.5h^2\ddot{y}\|_{\text{WRMS}} = 1$, where $\ddot{y}$ is an estimated second derivative of the solution at `t0`.

`CVodeSetMinStep`

Call `flag = CVodeSetMinStep(cvode_mem, hmin);`

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

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`hmin` (`realtype`) minimum absolute value of the step size (≥ 0.0).

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

`CV_SUCCESS` The optional value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CV_ILL_INPUT` Either `hmin` is nonpositive or it exceeds the maximum allowable step size.

Notes The default value is 0.0.

`CVodeSetMaxStep`

Call `flag = CVodeSetMaxStep(cvode_mem, hmax);`

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

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`hmax` (`realtype`) maximum absolute value of the step size (≥ 0.0).

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

`CV_SUCCESS` The optional value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CV_ILL_INPUT` Either `hmax` is nonpositive or it is smaller than the minimum allowable step size.

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

`CVodeSetStopTime`

Call `flag = CVodeSetStopTime(cvode_mem, tstop);`

Description The function `CVodeSetStopTime` specifies the value of the independent variable t past which the solution is not to proceed.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`tstop` (`realtype`) value of the independent variable past which the solution should not proceed.

Return value The return value `flag` (of type `int`) is one of
`CV_SUCCESS` The optional value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CV_ILL_INPUT` The value of `tstop` is not beyond the current t value, t_n.

Notes The default, if this routine is not called, is that no stop time is imposed.

`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 CVODES 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 CVODES 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`.

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 CVODES 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 `flag = CVodeSetNonlinConvCoef(cvode_mem, nlscoef);`

Description The function CVodeSetNonlinConvCoef specifies the safety factor used in the nonlinear convergence test (see §2.1).

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

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

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes The default value is 0.1.

CVodeSetIterType

Call `flag = CVodeSetIterType(cvode_mem, iter);`

Description The function CVodeSetIterType resets the nonlinear solver iteration type to iter.

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

iter (int) specifies the type of nonlinear solver iteration and may be either CV_NEWTON or CV_FUNCTIONAL.

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 iter value passed is neither CV_NEWTON nor CV_FUNCTIONAL.

Notes The nonlinear solver iteration type is initially specified in the call to CVodeCreate (see §4.5.1). This function call is needed only if iter is being changed from its value in the prior call to CVodeCreate.

CVodeSetConstraints

Call `flag = CVodeSetConstraints(cvode_mem, constraints);`

Description The function CVodeSetConstraints specifies a vector defining inequality constraints for each component of the solution vector y.

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

constraints (N_Vector) vector of constraint flags. If constraints[i] is

- 0.0 then no constraint is imposed on y_i.
- 1.0 then y_i will be constrained to be $y_i \geq 0.0$.
- −1.0 then y_i will be constrained to be $y_i \leq 0.0$.
- 2.0 then y_i will be constrained to be $y_i > 0.0$.
- −2.0 then y_i will be constrained to be $y_i < 0.0$.

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

CV_ILL_INPUT The constraints vector contains illegal values or the simultaneous corrector option has been selected when doing forward sensitivity analysis.

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

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

4.5.6.2 Direct linear solver interface optional input functions

The CVDLS solver interface needs a function to compute an approximation to the Jacobian matrix $J(t, y)$. This function must be of type `CVDlsJacFn`. 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 CVDLS solver. To specify a user-supplied Jacobian function `jac`, CVDLS provides the function `CVDlsSetJacFn`. The CVDLS 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`.

`CVDlsSetJacFn`

Call `flag = CVDlsSetJacFn(cvode_mem, jac);`

Description The function `CVDlsSetJacFn` specifies the Jacobian approximation function to be used.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`jac` (`CVDlsJacFn`) user-defined Jacobian approximation function.

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

`CVDLS_SUCCESS` The optional value has been successfully set.
`CVDLS_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CVDLS_LMEM_NULL` The CVDLS linear solver interface has not been initialized.

Notes By default, CVDLS uses an internal difference quotient function for dense and band matrices. If `NULL` is passed to `jac`, this default function is used. An error will occur if no `jac` is supplied when using a sparse matrix.

The function type `CVDlsJacFn` is described in §4.6.5.

4.5.6.3 Iterative linear solver interface optional input functions

If preconditioning is utilized with the CVSPILS linear solver interface, then the user must supply a preconditioner solve function `psolve` and specify its name in a call to `CVSpilsSetPreconditioner`. The evaluation and preprocessing of any Jacobian-related data needed by the user's preconditioner solve function is done in the optional user-supplied function `psetup`. Both of these functions are fully specified in §4.6. If used, the `psetup` function should also be specified in the call to `CVSpilsSetPreconditioner`.

The pointer `user_data` received through `CVodeSetUserData` (or a pointer to `NULL` if `user_data` was not specified) is passed to the preconditioner `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.

The CVSPILS 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 CVSPILS interface. A user-defined Jacobian-vector function must be of type `CVSpilsJacTimesVecFn` and can be specified through a call to `CVSpilsSetJacTimes` (see §4.6.6 for specification details). As with the user-supplied preconditioner functions, the evaluation and processing of any Jacobian-related data needed by the user's Jacobian-times-vector function is done in the optional user-supplied

function jtsetup (see §4.6.7 for specification details). As with the preconditioner functions, a pointer to the user-defined data structure, user_data, specified through CVodeSetUserData (or a NULL pointer otherwise) is passed to the Jacobian-times-vector setup and product functions, jtsetup and jtimes, each time they are called.

Finally, as described in Section 2.1, the CVSPILS interface requires that iterative linear solvers stop when the norm of the preconditioned residual is less than $0.05 \cdot (0.1\epsilon)$, where ϵ is the nonlinear solver tolerance. The user may adjust this linear solver tolerance by calling the function CVSpilsSetEpsLin.

CVSpilsSetPreconditioner

Call `flag = CVSpilsSetPreconditioner(cvode_mem, psetup, psolve);`

Description The function CVSpilsSetPreconditioner specifies the preconditioner setup and solve functions.

Arguments
- cvode_mem (void *) pointer to the CVODES memory block.
- psetup (CVSpilsPrecSetupFn) user-defined preconditioner setup function. Pass NULL if no setup is necessary.
- psolve (CVSpilsPrecSolveFn) user-defined preconditioner solve function.

Return value The return value flag (of type int) is one of
- CVSPILS_SUCCESS The optional values have been successfully set.
- CVSPILS_MEM_NULL The cvode_mem pointer is NULL.
- CVSPILS_LMEM_NULL The CVSPILS linear solver has not been initialized.
- CVSPILS_SUNLS_FAIL An error occurred when setting up preconditioning in the SUNLINSOL object used by the CVSPILS interface.

Notes The function type CVSpilsPrecSolveFn is described in §4.6.8. The function type CVSpilsPrecSetupFn is described in §4.6.9.

CVSpilsSetJacTimes

Call `flag = CVSpilsSetJacTimes(cvode_mem, jtsetup, jtimes);`

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

Arguments
- cvode_mem (void *) pointer to the CVODES memory block.
- jtsetup (CVSpilsJacTimesSetupFn) user-defined Jacobian-vector setup function. Pass NULL if no setup is necessary.
- jtimes (CVSpilsJacTimesVecFn) user-defined Jacobian-vector product function.

Return value The return value flag (of type int) is one of
- CVSPILS_SUCCESS The optional value has been successfully set.
- CVSPILS_MEM_NULL The cvode_mem pointer is NULL.
- CVSPILS_LMEM_NULL The CVSPILS linear solver has not been initialized.
- CVSPILS_SUNLS_FAIL An error occurred when setting up the system matrix-times-vector routines in the SUNLINSOL object used by the CVSPILS interface.

Notes By default, the CVSPILS linear solvers use an internal difference quotient function. If NULL is passed to jtimes, this default function is used.

The function type CVSpilsJacTimesSetupFn is described in §4.6.7.

The function type CVSpilsJacTimesVecFn is described in §4.6.6.

`CVSpilsSetEpsLin`

Call `flag = CVSpilsSetEpsLin(cvode_mem, eplifac);`

Description The function `CVSpilsSetEpsLin` specifies the factor by which the Krylov linear solver's convergence test constant is reduced from the Newton iteration test constant.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`eplifac` (`realtype`) linear convergence safety factor (≥ 0.0).

Return value The return value `flag` (of type `int`) is one of
`CVSPILS_SUCCESS` The optional value has been successfully set.
`CVSPILS_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CVSPILS_LMEM_NULL` The CVSPILS linear solver has not been initialized.
`CVSPILS_ILL_INPUT` The factor `eplifac` is negative.

Notes The default value is 0.05.
If `eplifac`$= 0.0$ is passed, the default value is used.

4.5.6.4 Rootfinding optional input functions

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

`CVodeSetRootDirection`

Call `flag = CVodeSetRootDirection(cvode_mem, rootdir);`

Description The function `CVodeSetRootDirection` specifies the direction of zero-crossings to be located and returned.

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

Return value The return value `flag` (of type `int`) is one of
`CV_SUCCESS` The optional value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CV_ILL_INPUT` rootfinding has not been activated through a call to `CVodeRootInit`.

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

`CVodeSetNoInactiveRootWarn`

Call `flag = CVodeSetNoInactiveRootWarn(cvode_mem);`

Description The function `CVodeSetNoInactiveRootWarn` disables issuing a warning if some root function appears to be identically zero at the beginning of the integration.

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

Return value The return value `flag` (of type `int`) is one of
`CV_SUCCESS` The optional value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.

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

4.5.7 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 CVODES.

The call to the `CVodeGetDky` function has the following form:

`CVodeGetDky`

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

Description The function `CVodeGetDky` computes the `k`-th derivative of the function `y` at time `t`, i.e. $d^{(k)}y/dt^{(k)}(t)$, where $t_n - h_u \le \mathtt{t} \le t_n$, t_n denotes the current internal time reached, and h_u is the last internal step size successfully used by the solver. The user may request `k` $= 0, 1, \ldots, q_u$, where q_u is the current order (optional output `qlast`).

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `t` (`realtype`) the value of the independent variable at which the derivative is to be evaluated.
- `k` (`int`) the derivative order requested.
- `dky` (`N_Vector`) vector containing the derivative. This vector must be allocated by the user.

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

`CV_SUCCESS`	`CVodeGetDky` succeeded.
`CV_BAD_K`	`k` is not in the range $0, 1, \ldots, q_u$.
`CV_BAD_T`	`t` is not in the interval $[t_n - h_u, t_n]$.
`CV_BAD_DKY`	The `dky` argument was `NULL`.
`CV_MEM_NULL`	The `cvode_mem` argument was `NULL`.

Notes It is only legal to call the function `CVodeGetDky` after a successful return from `CVode`. See `CVodeGetCurrentTime`, `CVodeGetLastOrder`, and `CVodeGetLastStep` in the next section for access to t_n, q_u, and h_u, respectively.

4.5.8 Optional output functions

CVODES provides an extensive set of functions that can be used to obtain solver performance information. Table 4.3 lists all optional output functions in CVODES, 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 CVODES 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 Newton iteration 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 direct 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.

4.5.8.1 SUNDIALS version information

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

Table 4.3: Optional outputs from CVODES, CVDLS, CVDIAG, and CVSPILS

Optional output	Function name
CVODES main solver	
Size of CVODES 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 CVODES integrator statistics	`CVodeGetIntegratorStats`
CVODES 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`
CVDLS linear solver interface	
Size of real and integer workspaces	`CVDlsGetWorkSpace`
No. of Jacobian evaluations	`CVDlsGetNumJacEvals`
No. of r.h.s. calls for finite diff. Jacobian evals.	`CVDlsGetNumRhsEvals`
Last return from a linear solver function	`CVDlsGetLastFlag`
Name of constant associated with a return flag	`CVDlsGetReturnFlagName`
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`
CVSPILS linear solver interface	
Size of real and integer workspaces	`CVSpilsGetWorkSpace`
No. of linear iterations	`CVSpilsGetNumLinIters`
No. of linear convergence failures	`CVSpilsGetNumConvFails`
No. of preconditioner evaluations	`CVSpilsGetNumPrecEvals`
No. of preconditioner solves	`CVSpilsGetNumPrecSolves`
No. of Jacobian-vector setup evaluations	`CVSpilsGetNumJTSetupEvals`
No. of Jacobian-vector product evaluations	`CVSpilsGetNumJtimesEvals`
No. of r.h.s. calls for finite diff. Jacobian-vector evals.	`CVSpilsGetNumRhsEvals`
Last return from a linear solver function	`CVSpilsGetLastFlag`
Name of constant associated with a return flag	`CVSpilsGetReturnFlagName`

`SUNDIALSGetVersion`

Call `flag = SUNDIALSGetVersion(version, len);`

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.

`SUNDIALSGetVersionNumber`

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

Description The function `SUNDIALSGetVersionNumber` set integers for the SUNDIALS major, minor, and patch release numbers and fills a character array with the release label if applicable.

Arguments `major` (`int`) SUNDIALS release major version number.
`minor` (`int`) SUNDIALS release minor version number.
`patch` (`int`) SUNDIALS release patch version number.
`label` (`char *`) character array to hold the SUNDIALS release label.
`len` (`int`) allocated length of the `label` character array.

Return value If successful, `SUNDIALSGetVersionNumber` returns 0 and the `major`, `minor`, `patch`, and `label` values are set. Otherwise, it returns -1 and the values are not set (the input character array is too short).

Notes A string of 10 characters should be sufficient to hold the label information. If a label is not used in the release version, no information is copied to `label`. Any trailing characters in the `label` array are removed.

4.5.8.2 Main solver optional output functions

CVODES 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 CVODES 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 CVODES nonlinear solver used. As a convenience, additional information extraction functions provide the optional outputs in groups. These optional output functions are described next.

`CVodeGetWorkSpace`

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

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

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`lenrw` (`long int`) the number of `realtype` values in the CVODES workspace.
`leniw` (`long int`) the number of integer values in the CVODES workspace.

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

`CV_SUCCESS` The optional output values have been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.

Notes In terms of the problem size N, the maximum method order `maxord`, and the number `nrtfn` of root functions (see §4.5.4), the actual size of the real workspace, in `realtype` words, is given by the following:

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

where N_r is the number of real words in one `N_Vector` ($\approx N$).

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

- base value: `leniw` $= 40 +$ (`maxord`+5) $* N_i$ $+$ `nrtfn`;
- using `CVodeSVtolerances`: `leniw` $=$ `leniw` $+N_i$;
- with constraint checking: `lenrw` $=$ `lenrw` $+N_i$;

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

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

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

Note that additional memory is allocated if quadratures and/or forward sensitivity integration is enabled. See §4.7.1 and §5.2.1 for more details.

`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 CVODES memory block.
`nsteps` (`long int`) number of steps taken by CVODES.

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 CVODES 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 CVODES 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`.

`CVodeGetNumErrTestFails`

Call `flag = CVodeGetNumErrTestFails(cvode_mem, &netfails);`

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

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`netfails` (`long int`) number of error test failures.

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

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

`CVodeGetLastOrder`

Call `flag = CVodeGetLastOrder(cvode_mem, &qlast);`

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

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`qlast` (`int`) method order used on the last internal step.

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

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

`CVodeGetCurrentOrder`

Call `flag = CVodeGetCurrentOrder(cvode_mem, &qcur);`

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

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`qcur` (`int`) method order to be used on the next internal step.

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

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

`CVodeGetLastStep`

Call `flag = CVodeGetLastStep(cvode_mem, &hlast);`

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

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`hlast` (`realtype`) step size taken on the last internal step.

Return value The return value `flag` (of type `int`) is one of
`CV_SUCCESS` The optional output value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`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 CVODES 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 CVODES 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 CVODES to ensure that the step size is within the prescribed bounds ($h_{\min} \le h_0 \le h_{\max}$), or to satisfy the local error test condition.

`CVodeGetCurrentTime`

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

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

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`tcur` (`realtype`) current internal time reached.

Return value The return value `flag` (of type `int`) is one of
`CV_SUCCESS` The optional output value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CVodeGetNumStabLimOrderReds`

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

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

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`nslred` (`long int`) number of order reductions due to stability limit detection.

Return value The return value `flag` (of type `int`) is one of
`CV_SUCCESS` The optional output value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.

Notes If the stability limit detection algorithm was not initialized (`CVodeSetStabLimDet` was not called), then `nslred` = 0.

`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 CVODES 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 CVODES 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 CVODES 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, togther 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, &tcur);
```

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

Arguments

`cvode_mem`	(`void *`) pointer to the CVODES memory block.
`nsteps`	(`long int`) number of steps taken by CVODES.
`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 (functional or Newton) iterations performed.

Arguments

`cvode_mem`	(`void *`) pointer to the CVODES 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`.

`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 CVODES 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 CVODES nonlinear solver statistics as a group.

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

`nniters` (`long int`) number of nonlinear iterations performed.

`nncfails` (`long int`) number of nonlinear convergence failures.

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

`CV_SUCCESS` The optional output value has been successfully set.

`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CVodeGetReturnFlagName`

Call `name = CVodeGetReturnFlagName(flag);`

Description The function `CVodeGetReturnFlagName` returns the name of the CVODES constant corresponding to `flag`.

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

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

4.5.8.3 Rootfinding optional output functions

There are two optional output functions associated with rootfinding.

`CVodeGetRootInfo`

Call `flag = CVodeGetRootInfo(cvode_mem, rootsfound);`

Description The function `CVodeGetRootInfo` returns an array showing which functions were found to have a root.

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

`rootsfound` (`int *`) array of length `nrtfn` with the indices of the user functions g_i found to have a root. For $i = 0, \ldots,$`nrtfn`-1, `rootsfound`$[i] \neq 0$ if g_i has a root, and $= 0$ if not.

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

`CV_SUCCESS` The optional output values have been successfully set.

`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.

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

The user must allocate memory for the vector `rootsfound`.

`CVodeGetNumGEvals`

Call `flag = CVodeGetNumGEvals(cvode_mem, &ngevals);`

Description The function `CVodeGetNumGEvals` returns the cumulative number of calls made to the user-supplied root function g.

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

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

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

`CV_SUCCESS` The optional output value has been successfully set.

`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.

4.5.8.4 Direct linear solver interface optional output functions

The following optional outputs are available from the CVDLS modules: workspace requirements, number of calls to the Jacobian routine, number of calls to the right-hand side routine for finite-difference Jacobian approximation, and last return value from a CVDLS 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`).

`CVDlsGetWorkSpace`

Call `flag = CVDlsGetWorkSpace(cvode_mem, &lenrwLS, &leniwLS);`

Description The function `CVDlsGetWorkSpace` returns the sizes of the real and integer workspaces used by the CVDLS linear solver interface.

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

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

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

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

`CVDLS_SUCCESS` The optional output values have been successfully set.

`CVDLS_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CVDLS_LMEM_NULL` The CVDLS linear solver has not been initialized.

Notes The workspace requirements reported by this routine correspond only to memory allocated within this interface and to memory allocated by the SUNLINSOL object attached to it. The template Jacobian matrix allocated by the user outside of CVDLS is not included in this report.

`CVDlsGetNumJacEvals`

Call `flag = CVDlsGetNumJacEvals(cvode_mem, &njevals);`

Description The function `CVDlsGetNumJacEvals` returns the number of calls made to the CVDLS Jacobian approximation function.

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

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

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

`CVDLS_SUCCESS` The optional output value has been successfully set.

`CVDLS_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CVDLS_LMEM_NULL` The CVDLS linear solver has not been initialized.

`CVDlsGetNumRhsEvals`

Call `flag = CVDlsGetNumRhsEvals(cvode_mem, &nfevalsLS);`

Description The function `CVDlsGetNumRhsEvals` returns the number of calls made to the user-supplied right-hand side function due to the finite difference Jacobian approximation.

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

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

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

`CVDLS_SUCCESS`	The optional output value has been successfully set.
`CVDLS_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CVDLS_LMEM_NULL`	The CVDLS linear solver has not been initialized.

Notes The value `nfevalsLS` is incremented only if one of the default internal difference quotient functions (dense or banded) is used.

`CVDlsGetLastFlag`

Call `flag = CVDlsGetLastFlag(cvode_mem, &lsflag);`

Description The function `CVDlsGetLastFlag` returns the last return value from a CVDLS routine.

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

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

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

`CVDLS_SUCCESS`	The optional output value has been successfully set.
`CVDLS_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CVDLS_LMEM_NULL`	The CVDLS linear solver has not been initialized.

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

`CVDlsGetReturnFlagName`

Call `name = CVDlsGetReturnFlagName(lsflag);`

Description The function `CVDlsGetReturnFlagName` returns the name of the CVDLS constant corresponding to `lsflag`.

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

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

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

4.5.8.5 Iterative linear solver interface optional output functions

The following optional outputs are available from the CVSPILS modules: workspace requirements, 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, number of calls to the right-hand side routine for finite-difference Jacobian-vector product approximation, 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`).

`CVSpilsGetWorkSpace`

Call `flag = CVSpilsGetWorkSpace(cvode_mem, &lenrwLS, &leniwLS);`

Description The function `CVSpilsGetWorkSpace` returns the global sizes of the CVSPILS real and integer workspaces.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`lenrwLS` (`long int`) the number of `realtype` values in the CVSPILS workspace.
`leniwLS` (`long int`) the number of integer values in the CVSPILS workspace.

Return value The return value `flag` (of type `int`) is one of
`CVSPILS_SUCCESS` The optional output value has been successfully set.
`CVSPILS_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CVSPILS_LMEM_NULL` The CVSPILS linear solver has not been initialized.

Notes The workspace requirements reported by this routine correspond only to memory allocated within this interface and to memory allocated by the SUNLINSOL object attached to it.

In a parallel setting, the above values are global (i.e., summed over all processors).

`CVSpilsGetNumLinIters`

Call `flag = CVSpilsGetNumLinIters(cvode_mem, &nliters);`

Description The function `CVSpilsGetNumLinIters` returns the cumulative number of linear iterations.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`nliters` (`long int`) the current number of linear iterations.

Return value The return value `flag` (of type `int`) is one of
`CVSPILS_SUCCESS` The optional output value has been successfully set.
`CVSPILS_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CVSPILS_LMEM_NULL` The CVSPILS linear solver has not been initialized.

`CVSpilsGetNumConvFails`

Call `flag = CVSpilsGetNumConvFails(cvode_mem, &nlcfails);`

Description The function `CVSpilsGetNumConvFails` returns the cumulative number of linear convergence failures.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`nlcfails` (`long int`) the current number of linear convergence failures.

Return value The return value `flag` (of type `int`) is one of
`CVSPILS_SUCCESS` The optional output value has been successfully set.
`CVSPILS_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CVSPILS_LMEM_NULL` The CVSPILS linear solver has not been initialized.

`CVSpilsGetNumPrecEvals`

Call `flag = CVSpilsGetNumPrecEvals(cvode_mem, &npevals);`

Description The function `CVSpilsGetNumPrecEvals` returns the number of preconditioner evaluations, i.e., the number of calls made to `psetup` with `jok = SUNFALSE`.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`npevals` (`long int`) the current number of calls to `psetup`.

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

`CVSPILS_SUCCESS`	The optional output value has been successfully set.
`CVSPILS_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CVSPILS_LMEM_NULL`	The CVSPILS linear solver has not been initialized.

`CVSpilsGetNumPrecSolves`

Call `flag = CVSpilsGetNumPrecSolves(cvode_mem, &npsolves);`

Description The function `CVSpilsGetNumPrecSolves` returns the cumulative number of calls made to the preconditioner solve function, `psolve`.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`npsolves` (`long int`) the current number of calls to `psolve`.

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

`CVSPILS_SUCCESS`	The optional output value has been successfully set.
`CVSPILS_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CVSPILS_LMEM_NULL`	The CVSPILS linear solver has not been initialized.

`CVSpilsGetNumJTSetupEvals`

Call `flag = CVSpilsGetNumJTSetupEvals(cvode_mem, &njtsetup);`

Description The function `CVSpilsGetNumJTSetupEvals` returns the cumulative number of calls made to the Jacobian-vector setup function `jtsetup`.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`njtsetup` (`long int`) the current number of calls to `jtsetup`.

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

`CVSPILS_SUCCESS`	The optional output value has been successfully set.
`CVSPILS_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CVSPILS_LMEM_NULL`	The CVSPILS linear solver has not been initialized.

`CVSpilsGetNumJtimesEvals`

Call `flag = CVSpilsGetNumJtimesEvals(cvode_mem, &njvevals);`

Description The function `CVSpilsGetNumJtimesEvals` returns the cumulative number of calls made to the Jacobian-vector function `jtimes`.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`njvevals` (`long int`) the current number of calls to `jtimes`.

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

`CVSPILS_SUCCESS`	The optional output value has been successfully set.
`CVSPILS_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CVSPILS_LMEM_NULL`	The CVSPILS linear solver has not been initialized.

`CVSpilsGetNumRhsEvals`

Call `flag = CVSpilsGetNumRhsEvals(cvode_mem, &nfevalsLS);`

Description The function `CVSpilsGetNumRhsEvals` returns the number of calls to the user right-hand side function for finite difference Jacobian-vector product approximation.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`nfevalsLS` (`long int`) the number of calls to the user right-hand side function.

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

`CVSPILS_SUCCESS` The optional output value has been successfully set.

`CVSPILS_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CVSPILS_LMEM_NULL` The CVSPILS linear solver has not been initialized.

Notes The value `nfevalsLS` is incremented only if the default `CVSpilsDQJtimes` difference quotient function is used.

`CVSpilsGetLastFlag`

Call `flag = CVSpilsGetLastFlag(cvode_mem, &lsflag);`

Description The function `CVSpilsGetLastFlag` returns the last return value from a CVSPILS routine.

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

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

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

`CVSPILS_SUCCESS` The optional output value has been successfully set.

`CVSPILS_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CVSPILS_LMEM_NULL` The CVSPILS linear solver has not been initialized.

Notes If the CVSPILS setup function failed (`CVode` returned `CV_LSETUP_FAIL`), `lsflag` will be `SUNLS_PSET_FAIL_UNREC`, `SUNLS_ASET_FAIL_UNREC`, or `SUNLS_PACKAGE_FAIL_UNREC`.

If the CVSPILS solve function failed (`CVode` returned `CV_LSOLVE_FAIL`), `lsflag` contains the error return flag from the SUNLINSOL object, which will be one of: `SUNLS_MEM_NULL`, indicating that the SUNLINSOL memory is `NULL`; `SUNLS_ATIMES_FAIL_UNREC`, indicating an unrecoverable failure in the $J*v$ function; `SUNLS_PSOLVE_FAIL_UNREC`, indicating that the preconditioner solve function `psolve` failed unrecoverably; `SUNLS_GS_FAIL`, indicating a failure in the Gram-Schmidt procedure (SPGMR and SPFGMR only); `SUNLS_QRSOL_FAIL`, indicating that the matrix R was found to be singular during the QR solve phase (SPGMR and SPFGMR only); or `SUNLS_PACKAGE_FAIL_UNREC`, indicating an unrecoverable failure in an external iterative linear solver package.

`CVSpilsGetReturnFlagName`

Call `name = CVSpilsGetReturnFlagName(lsflag);`

Description The function `CVSpilsGetReturnFlagName` returns the name of the CVSPILS constant corresponding to `lsflag`.

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

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

4.5.8.6 Diagonal linear solver interface optional output functions

The following optional outputs are available from the CVDIAG module: workspace requirements, number of calls to the right-hand side routine for finite-difference Jacobian approximation, and last return value from a CVDIAG function. Note that, where the name of an output would otherwise conflict with the name of an optional output from the main solver, a suffix `LS` (for Linear Solver) has been added here (e.g. `lenrwLS`).

`CVDiagGetWorkSpace`

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

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

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

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

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

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

CVDIAG_SUCCESS The optional output valus have been successfully set.

CVDIAG_MEM_NULL The cvode_mem pointer is NULL.

CVDIAG_LMEM_NULL The CVDIAG linear solver has not been initialized.

Notes In terms of the problem size N, the actual size of the real workspace is roughly $3N$ realtype words.

CVDiagGetNumRhsEvals

Call `flag = CVDiagGetNumRhsEvals(cvode_mem, &nfevalsLS);`

Description The function CVDiagGetNumRhsEvals returns the number of calls made to the user-supplied right-hand side function due to the finite difference Jacobian approximation.

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

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

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

CVDIAG_SUCCESS The optional output value has been successfully set.

CVDIAG_MEM_NULL The cvode_mem pointer is NULL.

CVDIAG_LMEM_NULL The CVDIAG linear solver has not been initialized.

Notes The number of diagonal approximate Jacobians formed is equal to the number of calls made to the linear solver setup function (see CVodeGetNumLinSolvSetups).

CVDiagGetLastFlag

Call `flag = CVDiagGetLastFlag(cvode_mem, &lsflag);`

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

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

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

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

CVDIAG_SUCCESS The optional output value has been successfully set.

CVDIAG_MEM_NULL The cvode_mem pointer is NULL.

CVDIAG_LMEM_NULL The CVDIAG linear solver has not been initialized.

Notes If the CVDIAG setup function failed (CVode returned CV_LSETUP_FAIL), the value of lsflag is equal to CVDIAG_INV_FAIL, indicating that a diagonal element with value zero was encountered. The same value is also returned if the CVDIAG solve function failed (CVode returned CV_LSOLVE_FAIL).

CVDiagGetReturnFlagName

Call `name = CVDiagGetReturnFlagName(lsflag);`

Description The function CVDiagGetReturnFlagName returns the name of the CVDIAG constant corresponding to lsflag.

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

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

4.5.9 CVODES reinitialization function

The function `CVodeReInit` reinitializes the main CVODES 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 CVDLS or CVSPILS interface routines, as described in §4.5.3. Otherwise, all solver inputs set previously remain in effect.

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

`CVodeReInit`

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

Description The function `CVodeReInit` provides required problem specifications and reinitializes CVODES.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `t0` (`realtype`) is the initial value of t.
- `y0` (`N_Vector`) is the initial value of y.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeReInit` was successful.
- `CV_MEM_NULL` The CVODES memory block was not initialized through a previous call to `CVodeCreate`.
- `CV_NO_MALLOC` Memory space for the CVODES memory block was not allocated through a previous call to `CVodeInit`.
- `CV_ILL_INPUT` An input argument to `CVodeReInit` has an illegal value.

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

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 (if Newton iteration is chosen), 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
```
typedef int (*CVRhsFn)(realtype t, N_Vector y, N_Vector ydot,
                       void *user_data);
```

Purpose This function computes the ODE right-hand side for a given value of the independent variable t and state vector y.

Arguments
- `t` is the current value of the independent variable.
- `y` is the current value of the dependent variable vector, $y(t)$.
- `ydot` is the output vector $f(t, y)$.
- `user_data` is the `user_data` pointer passed to `CVodeSetUserData`.

Return value A `CVRhsFn` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CV_RHSFUNC_FAIL` is returned).

Notes Allocation of memory for `ydot` is handled within CVODES.

A recoverable failure error return from the `CVRhsFn` is typically used to flag a value of the dependent variable `y` that is "illegal" in some way (e.g., negative where only a non-negative value is physically meaningful). If such a return is made, CVODES will attempt to recover (possibly repeating the Newton iteration, or reducing the step size) in order to avoid this recoverable error return.

For efficiency reasons, the right-hand side function is not evaluated at the converged solution of the nonlinear solver. Therefore, in general, a recoverable error in that converged value cannot be corrected. (It may be detected when the right-hand side function is called the first time during the following integration step, but a successful step cannot be undone.) However, if the user program also includes quadrature integration, the state variables can be checked for legality in the call to `CVQuadRhsFn`, which is called at the converged solution of the nonlinear system, and therefore CVODES can be flagged to attempt to recover from such a situation. Also, if sensitivity analysis is performed with one of the staggered methods, the ODE right-hand side function is called at the converged solution of the nonlinear system, and a recoverable error at that point can be flagged, and CVODES will then try to correct it.

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

4.6.2 Error message handler function

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

`CVErrHandlerFn`

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

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

Arguments
- `error_code` is the error code.
- `module` is the name of the CVODES module reporting the error.
- `function` is the name of the function in which the error occurred.
- `msg` is the error message.
- `eh_data` is a pointer to user data, the same as the `eh_data` parameter passed to `CVodeSetErrHandlerFn`.

Return value A `CVErrHandlerFn` function has no return value.

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

4.6.3 Error weight function

As an alternative to providing the relative and absolute tolerances, the user may provide a function of type `CVEwtFn` to compute a vector `ewt` containing the weights in the WRMS norm $\| v\|_{\text{WRMS}} = \sqrt{(1/N)\sum_1^N (W_i \cdot v_i)^2}$. These weights will be used in place of those defined by Eq. (2.7). The function type `CVEwtFn` is defined as follows:

`CVEwtFn`

Definition
```
typedef int (*CVEwtFn)(N_Vector y, N_Vector ewt, void *user_data);
```

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

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

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

Notes Allocation of memory for `ewt` is handled within CVODES.

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

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

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

Arguments
- `t` is the current value of the independent variable.
- `y` is the current value of the dependent variable vector, $y(t)$.
- `gout` is the output array, of length `nrtfn`, with components $g_i(t, y)$.
- `user_data` is a pointer to user data, the same as the `user_data` parameter passed to `CVodeSetUserData`.

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

Notes Allocation of memory for `gout` is automatically handled within CVODES.

4.6.5 Jacobian information (direct method Jacobian)

If the direct linear solver interface is used (i.e., `CVDlsSetLinearSolver` is called in the steps described in §4.4), the user may provide a function of type `CVDlsJacFn` defined as follows:

`CVDlsJacFn`

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

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

Arguments

`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 $f(t, y)$.
`Jac`	is the output Jacobian matrix (of type `SUNMatrix`).
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to `CVodeSetUserData`.
`tmp1`	
`tmp2`	
`tmp3`	are pointers to memory allocated for variables of type `N_Vector` which can be used by a `CVDlsJacFn` function as temporary storage or work space.

Return value A `CVDlsJacFn` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct, while CVDLS sets `last_flag` to `CVDLS_JACFUNC_RECVR`), or a negative value if it failed unrecoverably (in which case the integration is halted, `CVodes` returns `CV_LSETUP_FAIL` and CVDLS sets `last_flag` to `CVDLS_JACFUNC_UNRECVR`).

Notes Information regarding the structure of the specific SUNMATRIX structure (e.g. number of rows, upper/lower bandwidth, sparsity type) may be obtained through using the implementation-specific SUNMATRIX interface functions (see Chapter 8 for details).

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 `CVDlsJacFn` 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 §4.5.8.2. 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` $= 0 \ldots$ `N` $- 1$). This macro is meant for small problems for which efficiency of access is not a major concern. Thus, in terms of the indices m and n ranging from 1 to N, the Jacobian element $J_{m,n}$ can be set using the statement `SM_ELEMENT_D(J, m-1, n-1)` $= J_{m,n}$. Alternatively, `SM_COLUMN_D(J, j)` returns a pointer to the first element of the `j`-th column of `Jac` (with `j` $= 0 \ldots$ `N` $- 1$), and the elements of the `j`-th column can then be accessed using ordinary array indexing. Consequently, $J_{m,n}$ can be loaded using the statements `col_n = SM_COLUMN_D(J, n-1);` `col_n[m-1]` $= J_{m,n}$. For large problems, it is more efficient to use `SM_COLUMN_D` than to

use SM_ELEMENT_D. Note that both of these macros number rows and columns starting from 0. The SUNMATRIX_DENSE type and accessor macros are documented in §8.1.

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 -mupper $\leq$ m-n $\leq$ mlower. Alternatively, SM_COLUMN_B(J, j) returns a pointer to the diagonal element of the j-th column of Jac, and if we assign this address to realtype *col_j, then the i-th element of the j-th column is given by SM_COLUMN_ELEMENT_B(col_j, i, j), counting from 0. Thus, for (m, n) within the band, $J_{m,n}$ can be loaded by setting col_n = SM_COLUMN_B(J, n-1); SM_COLUMN_ELEMENT_B(col_n, m-1, n-1) = $J_{m,n}$. The elements of the j-th column can also be accessed via ordinary array indexing, but this approach requires knowledge of the underlying storage for a band matrix of type SUNMATRIX_BAND. The array col_n can be indexed from $-$mupper to mlower. For large problems, it is more efficient to use SM_COLUMN_B and SM_COLUMN_ELEMENT_B than to use the SM_ELEMENT_B macro. As in the dense case, these macros all number rows and columns starting from 0. The SUNMATRIX_BAND type and accessor macros are documented in §8.2.

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 §8.3.

4.6.6 Jacobian information (matrix-vector product)

If the CVSPILS solver interface is selected (i.e., CVSpilsSetLinearSolver is called in the steps described in §4.4), the user may provide a function of type CVSpilsJacTimesVecFn 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.

CVSpilsJacTimesVecFn

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

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

Arguments

v	is the vector by which the Jacobian must be multiplied.
Jv	is the output vector computed.
t	is the current value of the independent variable.
y	is the current value of the dependent variable vector.

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

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

`tmp` is a pointer to memory allocated for a variable of type `N_Vector` which can be used for work space.

Return value The value returned by the Jacobian-vector product function should be 0 if successful. Any other return value will result in an unrecoverable error of the generic Krylov solver, in which case the integration is halted.

Notes This function must return a value of $J * v$ that uses the *current* value of J, i.e. as evaluated at the current (t, y).
If the user's `CVSpilsJacTimesVecFn` function uses difference quotient approximations, it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to `cv_mem` to `user_data` and then use the `CVodeGet*` functions described in §4.5.8.2. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

4.6.7 Jacobian information (matrix-vector setup)

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

`CVSpilsJacTimesSetupFn`

Definition
```
typedef int (*CVSpilsJacTimesSetupFn)(realtype t, N_Vector y,
                                      N_Vector fy, void *user_data);
```

Purpose This function preprocesses and/or evaluates Jacobian-related data needed by the Jacobian-times-vector routine.

Arguments `t` is the current value of the independent variable.

`y` is the current value of the dependent variable vector.

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

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

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

Notes Each call to the Jacobian-vector setup function is preceded by a call to the `CVRhsFn` user function with the same `(t,y)` arguments. Thus, the setup function can use any auxiliary data that is computed and saved during the evaluation of the ODE right-hand side.

If the user's `CVSpilsJacTimesSetupFn` function uses difference quotient approximations, it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to `cv_mem` to `user_data` and then use the `CVodeGet*` functions described in §4.5.8.2. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

4.6.8 Preconditioning (linear system solution)

If preconditioning is used, 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 Newton 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 CVSpilsPrecSolveFn, defined as follows:

CVSpilsPrecSolveFn

Definition
```
typedef int (*CVSpilsPrecSolveFn)(realtype t, N_Vector y, N_Vector fy,
                                  N_Vector r, N_Vector z, realtype gamma,
                                  realtype delta, int lr, void *user_data);
```

Purpose This function solves the preconditioned system $Pz = r$.

Arguments

t	is the current value of the independent variable.
y	is the current value of the dependent variable vector.
fy	is the current value of the vector $f(t, y)$.
r	is the right-hand side vector of the linear system.
z	is the computed output vector.
gamma	is the scalar γ appearing in the Newton matrix given by $M = I - \gamma J$.
delta	is an input tolerance to be used if an iterative method is employed in the solution. In that case, the residual vector $Res = r - Pz$ of the system should be made less than delta in the weighted l_2 norm, i.e., $\sqrt{\sum_i (Res_i \cdot ewt_i)^2} <$ delta. To obtain the N_Vector ewt, call CVodeGetErrWeights (see §4.5.8.2).
lr	is an input flag indicating whether the preconditioner solve function is to use the left preconditioner (lr = 1) or the right preconditioner (lr = 2);
user_data	is a pointer to user data, the same as the user_data parameter passed to the function CVodeSetUserData.

Return value The value returned by the preconditioner solve function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

4.6.9 Preconditioning (Jacobian data)

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 CVSpilsPrecSetupFn, defined as follows:

CVSpilsPrecSetupFn

Definition
```
typedef int (*CVSpilsPrecSetupFn)(realtype t, N_Vector y, N_Vector fy,
                                  booleantype jok, booleantype *jcurPtr,
                                  realtype gamma, void *user_data);
```

Purpose This function preprocesses and/or evaluates Jacobian-related data needed by the preconditioner.

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 Newton matrix $M = I - \gamma J$.

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

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 The operations performed by this function might include forming a crude approximate Jacobian and performing an LU factorization of the resulting approximation to $M = I - \gamma J$.

Each call to the preconditioner setup function is preceded by a call to the `CVRhsFn` user function with the same `(t,y)` arguments. Thus, the preconditioner setup function can use any auxiliary data that is computed and saved during the evaluation of the ODE right-hand side.

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 Newton iteration.

If the user's `CVSpilsPrecSetupFn` 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 §4.5.8.2. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

4.7 Integration of pure quadrature equations

CVODES allows the ODE system to include *pure quadratures*. In this case, it is more efficient to treat the quadratures separately by excluding them from the nonlinear solution stage. To do this, begin by excluding the quadrature variables from the vector `y` and excluding the quadrature equations from within `res`. Thus a separate vector `yQ` of quadrature variables is to satisfy $(d/dt)\texttt{yQ} = f_Q(t, y)$. The following is an overview of the sequence of calls in a user's main program in this situation. Steps that are unchanged from the skeleton program presented in §4.4 are grayed out.

1. **Initialize parallel or multi-threaded environment, if appropriate**
2. **Set problem dimensions, etc.**

   Set the problem size `N` (excluding quadrature variables), and the number of quadrature variables `Nq`.

   If appropriate, set the local vector length `Nlocal` (excluding quadrature variables), and the local number of quadrature variables `Nqlocal`.
3. **Set vector of initial values**
4. **Create CVODES object**
5. **Allocate internal memory**
6. **Set optional inputs**
7. **Attach linear solver module**

8. **Set linear solver optional inputs**

9. **Set vector `yQ0` of initial values for quadrature variables**

   Typically, the quadrature variables should be initialized to 0.

10. **Initialize quadrature integration**

    Call `CVodeQuadInit` to specify the quadrature equation right-hand side function and to allocate internal memory related to quadrature integration. See §4.7.1 for details.

11. **Set optional inputs for quadrature integration**

    Call `CVodeSetQuadErrCon` to indicate whether or not quadrature variables shoule be used in the step size control mechanism, and to specify the integration tolerances for quadrature variables. See §4.7.4 for details.

12. **Advance solution in time**

13. **Extract quadrature variables**

    Call `CVodeGetQuad` to obtain the values of the quadrature variables at the current time. See §4.7.3 for details.

14. **Get optional outputs**

15. **Get quadrature optional outputs**

    Call `CVodeGetQuad*` functions to obtain optional output related to the integration of quadratures. See §4.7.5 for details.

16. **Deallocate memory for solution vector and for the vector of quadrature variables**

17. **Free solver memory**

18. **Finalize MPI, if used**

`CVodeQuadInit` can be called and quadrature-related optional inputs (step 11 above) can be set anywhere between steps 4 and 12.

4.7.1 Quadrature initialization and deallocation functions

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

`CVodeQuadInit`

Call `flag = CVodeQuadInit(cvode_mem, fQ, yQ0);`

Description The function `CVodeQuadInit` provides required problem specifications, allocates internal memory, and initializes quadrature integration.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block returned by `CVodeCreate`.
- `fQ` (`CVQuadRhsFn`) is the C function which computes f_Q, the right-hand side of the quadrature equations. This function has the form `fQ(t, y, yQdot, fQ_data)` (for full details see §4.7.6).
- `yQ0` (`N_Vector`) is the initial value of `yQ`.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeQuadInit` was successful.
- `CV_MEM_NULL` The CVODES memory was not initialized by a prior call to `CVodeCreate`.
- `CV_MEM_FAIL` A memory allocation request failed.

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

In terms of the number of quadrature variables N_q and maximum method order maxord, the size of the real workspace is increased as follows:

- Base value: lenrw = lenrw + (maxord+5)N_q
- If using CVodeSVtolerances (see CVodeSetQuadErrCon): lenrw = lenrw +N_q

the size of the integer workspace is increased as follows:

- Base value: leniw = leniw + (maxord+5)N_q
- If using CVodeSVtolerances: leniw = leniw +N_q

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

CVodeQuadReInit

Call `flag = CVodeQuadReInit(cvode_mem, yQ0);`

Description The function CVodeQuadReInit provides required problem specifications and reinitializes the quadrature integration.

Arguments
- cvode_mem (void *) pointer to the CVODES memory block.
- yQ0 (N_Vector) is the initial value of yQ.

Return value The return value flag (of type int) will be one of the following:
- CV_SUCCESS The call to CVodeReInit was successful.
- CV_MEM_NULL The CVODES memory was not initialized by a prior call to CVodeCreate.
- CV_NO_QUAD Memory space for the quadrature integration was not allocated by a prior call to CVodeQuadInit.

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

CVodeQuadFree

Call `CVodeQuadFree(cvode_mem);`

Description The function CVodeQuadFree frees the memory allocated for quadrature integration.

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

Return value The function CVodeQuadFree has no return value.

Notes In general, CVodeQuadFree need not be called by the user as it is invoked automatically by CVodeFree.

4.7.2 CVODES solver function

Even if quadrature integration was enabled, the call to the main solver function CVode is exactly the same as in §4.5.5. However, in this case the return value flag can also be one of the following:

CV_QRHSFUNC_FAIL The quadrature right-hand side function failed in an unrecoverable manner.

CV_FIRST_QRHSFUNC_FAIL The quadrature right-hand side function failed at the first call.

CV_REPTD_QRHSFUNC_ERR	Convergence test failures occurred too many times due to repeated recoverable errors in the quadrature right-hand side function. This value will also be returned if the quadrature right-hand side function had repeated recoverable errors during the estimation of an initial step size (assuming the quadrature variables are included in the error tests).
CV_UNREC_RHSFUNC_ERR	The quadrature right-hand function had a recoverable error, but no recovery was possible. This failure mode is rare, as it can occur only if the quadrature right-hand side function fails recoverably after an error test failed while at order one.

4.7.3 Quadrature extraction functions

If quadrature integration has been initialized by a call to CVodeQuadInit, or reinitialized by a call to CVodeQuadReInit, then CVODES computes both a solution and quadratures at time t. However, CVode will still return only the solution y in yout. Solution quadratures can be obtained using the following function:

CVodeGetQuad

Call `flag = CVodeGetQuad(cvode_mem, &tret, yQ);`

Description The function CVodeGetQuad returns the quadrature solution vector after a successful return from CVode.

Arguments

cvode_mem	(void *) pointer to the memory previously allocated by CVodeInit.
tret	(realtype) the time reached by the solver (output).
yQ	(N_Vector) the computed quadrature vector.

Return value The return value flag of CVodeGetQuad is one of:

CV_SUCCESS	CVodeGetQuad was successful.
CV_MEM_NULL	cvode_mem was NULL.
CV_NO_QUAD	Quadrature integration was not initialized.
CV_BAD_DKY	yQ is NULL.

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

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

CVodeGetQuadDky

Call `flag = CVodeGetQuadDky(cvode_mem, t, k, dkyQ);`

Description The function CVodeGetQuadDky returns derivatives of the quadrature solution vector after a successful return from CVode.

Arguments

cvode_mem	(void *) pointer to the memory previously allocated by CVodeInit.
t	(realtype) the time at which quadrature information is requested. The time t must fall within the interval defined by the last successful step taken by CVODES.
k	(int) order of the requested derivative. This must be $\leq$ qlast.
dkyQ	(N_Vector) the vector containing the derivative. This vector must be allocated by the user.

Return value The return value flag of CVodeGetQuadDky is one of:

CV_SUCCESS	CVodeGetQuadDky succeeded.
CV_MEM_NULL	The pointer to cvode_mem was NULL.

CV_NO_QUAD	Quadrature integration was not initialized.
CV_BAD_DKY	The vector dkyQ is NULL.
CV_BAD_K	k is not in the range $0, 1, \ldots,$ qlast.
CV_BAD_T	The time t is not in the allowed range.

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

4.7.4 Optional inputs for quadrature integration

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

CVodeSetQuadErrCon

Call `flag = CVodeSetQuadErrCon(cvode_mem, errconQ);`

Description The function CVodeSetQuadErrCon specifies whether or not the quadrature variables are to be used in the step size control mechanism within CVODES. If they are, the user must call CVodeQuadSStolerances or CVodeQuadSVtolerances to specify the integration tolerances for the quadrature variables.

Arguments

cvode_mem	(void *) pointer to the CVODES memory block.
errconQ	(booleantype) specifies whether quadrature variables are included (SUNTRUE) or not (SUNFALSE) in the error control mechanism.

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_NO_QUAD	Quadrature integration has not been initialized.

Notes By default, errconQ is set to SUNFALSE.

It is illegal to call CVodeSetQuadErrCon before a call to CVodeQuadInit.

If the quadrature variables are part of the step size control mechanism, one of the following functions must be called to specify the integration tolerances for quadrature variables.

CVodeQuadSStolerances

Call `flag = CVodeQuadSVtolerances(cvode_mem, reltolQ, abstolQ);`

Description The function CVodeQuadSStolerances specifies scalar relative and absolute tolerances.

Arguments

cvode_mem	(void *) pointer to the CVODES memory block.
reltolQ	(realtype) is the scalar relative error tolerance.
abstolQ	(realtype) is the scalar absolute error tolerance.

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

CV_SUCCESS	The optional value has been successfully set.
CV_NO_QUAD	Quadrature integration was not initialized.
CV_MEM_NULL	The cvode_mem pointer is NULL.
CV_ILL_INPUT	One of the input tolerances was negative.

CVodeQuadSVtolerances

Call `flag = CVodeQuadSVtolerances(cvode_mem, reltolQ, abstolQ);`

Description The function CVodeQuadSVtolerances specifies scalar relative and vector absolute tolerances.

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

`reltolQ` (`realtype`) is the scalar relative error tolerance.
`abstolQ` (`N_Vector`) is the vector absolute error tolerance.

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

`CV_SUCCESS` The optional value has been successfully set.
`CV_NO_QUAD` Quadrature integration was not initialized.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CV_ILL_INPUT` One of the input tolerances was negative.

4.7.5 Optional outputs for quadrature integration

CVODES provides the following functions that can be used to obtain solver performance information related to quadrature integration.

`CVodeGetQuadNumRhsEvals`

Call `flag = CVodeGetQuadNumRhsEvals(cvode_mem, &nfQevals);`

Description The function `CVodeGetQuadNumRhsEvals` returns the number of calls made to the user's quadrature right-hand side function.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`nfQevals` (`long int`) number of calls made to the user's `fQ` function.

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

`CV_SUCCESS` The optional output value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CV_NO_QUAD` Quadrature integration has not been initialized.

`CVodeGetQuadNumErrTestFails`

Call `flag = CVodeGetQuadNumErrTestFails(cvode_mem, &nQetfails);`

Description The function `CVodeGetQuadNumErrTestFails` returns the number of local error test failures due to quadrature variables.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`nQetfails` (`long int`) number of error test failures due to quadrature variables.

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

`CV_SUCCESS` The optional output value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CV_NO_QUAD` Quadrature integration has not been initialized.

`CVodeGetQuadErrWeights`

Call `flag = CVodeGetQuadErrWeights(cvode_mem, eQweight);`

Description The function `CVodeGetQuadErrWeights` returns the quadrature error weights at the current time.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`eQweight` (`N_Vector`) quadrature 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`.
`CV_NO_QUAD` Quadrature integration has not been initialized.

Notes The user must allocate memory for `eQweight`.

If quadratures were not included in the error control mechanism (through a call to `CVodeSetQuadErrCon` with `errconQ = SUNTRUE`), `CVodeGetQuadErrWeights` does not set the `eQweight` vector.

`CVodeGetQuadStats`

Call `flag = CVodeGetQuadStats(cvode_mem, &nfQevals, &nQetfails);`

Description The function `CVodeGetQuadStats` returns the CVODES integrator statistics as a group.

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

`nfQevals` (`long int`) number of calls to the user's `fQ` function.

`nQetfails` (`long int`) number of error test failures due to quadrature variables.

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_NO_QUAD` Quadrature integration has not been initialized.

4.7.6 User-supplied function for quadrature integration

For integration of quadrature equations, the user must provide a function that defines the right-hand side of the quadrature equations (in other words, the integrand function of the integral that must be evaluated). This function must be of type `CVQuadRhsFn` defined as follows:

`CVQuadRhsFn`

Definition
```
typedef int (*CVQuadRhsFn)(realtype t, N_Vector y,
                           N_Vector yQdot, void *user_data);
```

Purpose This function computes the quadrature equation right-hand side for a given value of the independent variable t and state vector y.

Arguments
`t` is the current value of the independent variable.

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

`yQdot` is the output vector $f_Q(t, y)$.

`user_data` is the `user_data` pointer passed to `CVodeSetUserData`.

Return value A `CVQuadRhsFn` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CV_QRHSFUNC_FAIL` is returned).

Notes Allocation of memory for `yQdot` is automatically handled within CVODES.

Both `y` and `yQdot` are of type `N_Vector`, but they typically have different internal representations. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with CVODES do not perform any consistency checks with respect to their `N_Vector` arguments (see §7.1 and §7.2).

There are two situations in which recovery is not possible even if `CVQuadRhsFn` function returns a recoverable error flag. One is when this occurs at the very first call to the `CVQuadRhsFn` (in which case CVODES returns `CV_FIRST_QRHSFUNC_ERR`). The other is when a recoverable error is reported by `CVQuadRhsFn` after an error test failure, while the linear multistep method order is equal to 1 (in which case CVODES returns `CV_UNREC_QRHSFUNC_ERR`).

4.8 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, CVODES provides a banded preconditioner in the module CVBANDPRE and a band-block-diagonal preconditioner module CVBBDPRE.

4.8.1 A serial banded preconditioner module

This preconditioner provides a band matrix preconditioner for use with the CVSPILS iterative 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 `cvodes_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**
3. **Set vector of initial values**
4. **Create CVODES object**
5. **Initialize CVODES solver**
6. **Specify integration tolerances**
7. **Set optional inputs**
8. **Create linear solver object**

   When creating the iterative linear solver object, specify the type of preconditioning (`PREC_LEFT` or `PREC_RIGHT`) to use.
9. **Set linear solver optional inputs**
10. **Attach linear solver module**
11. **Initialize the CVBANDPRE preconditioner module**

    Specify the upper and lower half-bandwidths (`mu` and `ml`, respectively) and call

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

    to allocate memory and initialize the internal preconditioner data.
12. **Set linear solver interface optional inputs**

    Note that the user should not overwrite the preconditioner setup function or solve function through calls to the `CVSpilsSetPreconditioner` optional input function.
13. **Specify rootfinding problem**

14. **Advance solution in time**

15. **Get optional outputs**

    Additional optional outputs associated with CVBANDPRE are available by way of two routines described below, `CVBandPrecGetWorkSpace` and `CVBandPrecGetNumRhsEvals`.

16. **Deallocate memory for solution vector**

17. **Free solver memory**

18. **Free linear solver memory**

The CVBANDPRE preconditioner module is initialized and attached by calling the following function:

`CVBandPrecInit`

Call `flag = CVBandPrecInit(cvode_mem, N, mu, ml);`

Description The function `CVBandPrecInit` initializes the CVBANDPRE preconditioner and allocates required (internal) memory for it.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `N` (`sunindextype`) problem dimension.
- `mu` (`sunindextype`) upper half-bandwidth of the Jacobian approximation.
- `ml` (`sunindextype`) lower half-bandwidth of the Jacobian approximation.

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

`CVSPILS_SUCCESS`	The call to `CVBandPrecInit` was successful.
`CVSPILS_MEM_NULL`	The `cvode_mem` pointer was `NULL`.
`CVSPILS_MEM_FAIL`	A memory allocation request has failed.
`CVSPILS_LMEM_NULL`	A CVSPILS linear solver memory was not attached.
`CVSPILS_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 $-\texttt{ml} \leq j - i \leq \texttt{mu}$.

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

`CVBandPrecGetWorkSpace`

Call `flag = CVBandPrecGetWorkSpace(cvode_mem, &lenrwBP, &leniwBP);`

Description The function `CVBandPrecGetWorkSpace` returns the sizes of the CVBANDPRE real and integer workspaces.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `lenrwBP` (`long int`) the number of `realtype` values in the CVBANDPRE workspace.
- `leniwBP` (`long int`) the number of integer values in the CVBANDPRE workspace.

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

`CVSPILS_SUCCESS`	The optional output values have been successfully set.
`CVSPILS_PMEM_NULL`	The CVBANDPRE preconditioner has not been initialized.

Notes The workspace requirements reported by this routine correspond only to memory allocated within the CVBANDPRE module (the banded matrix approximation, banded SUNLINSOL object, and temporary vectors).

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

`CVBandPrecGetNumRhsEvals`

Call `flag = CVBandPrecGetNumRhsEvals(cvode_mem, &nfevalsBP);`

Description The function `CVBandPrecGetNumRhsEvals` returns the number of calls made to the user-supplied right-hand side function for the finite difference banded Jacobian approximation used within the preconditioner setup function.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`nfevalsBP` (`long int`) the number of calls to the user right-hand side function.

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

`CVSPILS_SUCCESS`	The optional output value has been successfully set.
`CVSPILS_PMEM_NULL`	The CVBANDPRE preconditioner has not been initialized.

Notes The counter `nfevalsBP` is distinct from the counter `nfevalsLS` returned by the corresponding function `CVSpilsGetNumRhsEvals` and `nfevals` returned by `CVodeGetNumRhsEvals`. The total number of right-hand side function evaluations is the sum of all three of these counters.

4.8.2 A parallel band-block-diagonal preconditioner module

A principal reason for using a parallel ODE solver such as CVODES 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 [24] and is included in a software module within the CVODES package. This module works with the parallel vector module NVECTOR_PARALLEL and is usable with any of the Krylov iterative linear solvers through the CVSPILS 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

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

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

$$P = diag[P_1, P_2, \ldots, P_M] \tag{4.2}$$

where

$$P_m \approx I - \gamma J_m \tag{4.3}$$

and J_m is a difference quotient approximation to $\partial g_m/\partial y_m$. This matrix is taken to be banded, with upper and lower half-bandwidths `mudq` and `mldq` defined as the number of non-zero diagonals above and below the main diagonal, respectively. The difference quotient approximation is computed using `mudq` + `mldq` +2 evaluations of g_m, but only a matrix of bandwidth `mukeep` + `mlkeep` +1 is retained. Neither pair of parameters need be the true half-bandwidths of the Jacobian of the local block of g, if smaller values provide a more efficient preconditioner. The solution of the complete linear system

$$Px = b \tag{4.4}$$

reduces to solving each of the equations

$$P_m x_m = b_m \tag{4.5}$$

and this is done by banded LU factorization of P_m followed by a banded backsolve.

Similar block-diagonal preconditioners could be considered with different treatments of the blocks P_m. For example, incomplete LU factorization or an iterative method could be used instead of banded LU factorization.

The CVBBDPRE module calls two user-provided functions to construct P: a required function `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
```
typedef int (*CVLocalFn)(sunindextype Nlocal, realtype t, N_Vector y,
                         N_Vector glocal, void *user_data);
```

Purpose This `gloc` function computes $g(t, y)$. It loads the vector `glocal` as a function of `t` and `y`.

Arguments
- `Nlocal` is the local vector length.
- `t` is the value of the independent variable.
- `y` is the dependent variable.
- `glocal` is the output vector.
- `user_data` is a pointer to user data, the same as the `user_data` parameter passed to `CVodeSetUserData`.

Return value A `CVLocalFn` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CVode` returns `CV_LSETUP_FAIL`).

Notes This function must assume that all interprocess communication of data needed to calculate `glocal` has already been done, and that this data is accessible within `user_data`.

The case where g is mathematically identical to f is allowed.

`CVCommFn`

Definition
```
typedef int (*CVCommFn)(sunindextype Nlocal, realtype t,
                        N_Vector y, void *user_data);
```

Purpose This `cfn` function performs all interprocess communication necessary for the execution of the `gloc` function above, using the input vector `y`.

Arguments
- `Nlocal` is the local vector length.
- `t` is the value of the independent variable.

`y` is the dependent variable.

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

Return value A `CVCommFn` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CVode` returns `CV_LSETUP_FAIL`).

Notes The `cfn` function is expected to save communicated data in space defined within the data structure `user_data`.

Each call to the `cfn` function is preceded by a call to the right-hand side function `f` with the same (`t`, `y`) arguments. Thus, `cfn` can omit any communication done by `f` if relevant to the evaluation of `glocal`. If all necessary communication was done in `f`, then `cfn = NULL` can be passed in the call to `CVBBDPrecInit` (see below).

Besides the header files required for the integration of the ODE problem (see §4.3), to use the CVBBDPRE module, the main program must include the header file `cvodes_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**
3. **Set vector of initial values**
4. **Create CVODES object**
5. **Initialize CVODES solver**
6. **Specify integration tolerances**
7. **Set optional inputs**
8. **Create linear solver object**

   When creating the iterative linear solver object, specify the type of preconditioning (`PREC_LEFT` or `PREC_RIGHT`) to use.
9. **Set linear solver optional inputs**
10. **Attach linear solver module**
11. **Initialize the CVBBDPRE preconditioner module**

    Specify the upper and lower half-bandwidths `mudq` and `mldq`, and `mukeep` and `mlkeep`, and call

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

    to allocate memory and initialize the internal preconditioner data. The last two arguments of `CVBBDPrecInit` are the two user-supplied functions described above.
12. **Set linear solver interface optional inputs**

    Note that the user should not overwrite the preconditioner setup function or solve function through calls to the `CVSpilsSetPreconditioner` optional input function.
13. **Advance solution in time**

14. **Get optional outputs**

    Additional optional outputs associated with CVBBDPRE are available by way of two routines described below, `CVBBDPrecGetWorkSpace` and `CVBBDPrecGetNumGfnEvals`.

15. **Deallocate memory for solution vector**

16. **Free solver memory**

17. **Free linear solver memory**

18. **Finalize MPI**

The user-callable functions that initialize (step 11 above) or re-initialize the CVBBDPRE preconditioner module are described next.

`CVBBDPrecInit`

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

Description The function `CVBBDPrecInit` initializes and allocates (internal) memory for the CVBBDPRE preconditioner.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `local_N` (`sunindextype`) local vector length.
- `mudq` (`sunindextype`) upper half-bandwidth to be used in the difference quotient Jacobian approximation.
- `mldq` (`sunindextype`) lower half-bandwidth to be used in the difference quotient Jacobian approximation.
- `mukeep` (`sunindextype`) upper half-bandwidth of the retained banded approximate Jacobian block.
- `mlkeep` (`sunindextype`) lower half-bandwidth of the retained banded approximate Jacobian block.
- `dqrely` (`realtype`) the relative increment in components of `y` used in the difference quotient approximations. The default is `dqrely`$= \sqrt{\text{unit roundoff}}$, which can be specified by passing `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
- `CVSPILS_SUCCESS` The call to `CVBBDPrecInit` was successful.
- `CVSPILS_MEM_NULL` The `cvode_mem` pointer was `NULL`.
- `CVSPILS_MEM_FAIL` A memory allocation request has failed.
- `CVSPILS_LMEM_NULL` A CVSPILS linear solver was not attached.
- `CVSPILS_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 CVODES 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 `CVSpilsSet***` 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 CVODES 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 `dqrely` $= \sqrt{\text{unit roundoff}}$, which can be specified by passing `dqrely` $= 0.0$.

Return value The return value `flag` (of type `int`) is one of
- `CVSPILS_SUCCESS` The call to `CVBBDPrecReInit` was successful.
- `CVSPILS_MEM_NULL` The `cvode_mem` pointer was `NULL`.
- `CVSPILS_LMEM_NULL` A CVSPILS linear solver memory was not attached.
- `CVSPILS_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 CVODES 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
- `CVSPILS_SUCCESS` The optional output value has been successfully set.
- `CVSPILS_MEM_NULL` The `cvode_mem` pointer was `NULL`.
- `CVSPILS_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 `CVSpilsGetWorkSpace`.

`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 CVODES 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
- `CVSPILS_SUCCESS` The optional output value has been successfully set.
- `CVSPILS_MEM_NULL` The `cvode_mem` pointer was `NULL`.
- `CVSPILS_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 CVODES output and `npsolves` and `nfevalsLS` are linear solver optional outputs (see §4.5.8).

Chapter 5

Using CVODES for Forward Sensitivity Analysis

This chapter describes the use of CVODES to compute solution sensitivities using forward sensitivity analysis. One of our main guiding principles was to design the CVODES user interface for forward sensitivity analysis as an extension of that for IVP integration. Assuming a user main program and user-defined support routines for IVP integration have already been defined, in order to perform forward sensitivity analysis the user only has to insert a few more calls into the main program and (optionally) define an additional routine which computes the right-hand side of the sensitivity systems (2.11). The only departure from this philosophy is due to the `CVRhsFn` type definition (§4.6.1). Without changing the definition of this type, the only way to pass values of the problem parameters to the ODE right-hand side function is to require the user data structure `f_data` to contain a pointer to the array of real parameters p.

CVODES 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.

We begin with a brief overview, in the form of a skeleton user program. Following that are detailed descriptions of the interface to the various user-callable routines and of the user-supplied routines that were not already described in Chapter 4.

5.1 A skeleton of the user's main program

The following is a skeleton of the user's main program (or calling program) as an application of CVODES. The user program is to have these steps in the order indicated, unless otherwise noted. For the sake of brevity, we defer many of the details to the later sections. As in §4.4, most steps are independent of the NVECTOR implementation used; where this is not the case, refer to Chapter 7 for specifics. Differences between the user main program in §4.4 and the one below start only at step (13). Steps that are unchanged from the skeleton program presented in §4.4 are grayed out.

First, note that no additional header files need be included for forward sensitivity analysis beyond those for IVP solution (§4.4).

1. **Initialize parallel or multi-threaded environment, if appropriate**
2. **Set problem dimensions etc.**
3. **Set vector of initial values**
4. **Create CVODES object**
5. **Initialize CVODES**
6. **Specify integration tolerances**

7. **Set optional inputs**
8. **Create matrix object**
9. **Create linear solver object**
10. **Set linear solver optional inputs**
11. **Attach linear solver module**
12. **Initialize quadrature problem, if not sensitivity-dependent**
13. **Define the sensitivity problem**

    •Number of sensitivities (required)

    Set `Ns` $= N_s$, the number of parameters with respect to which sensitivities are to be computed.

    •Problem parameters (optional)

    If CVODES is to evaluate the right-hand sides of the sensitivity systems, set `p`, an array of `Np` real parameters upon which the IVP depends. Only parameters with respect to which sensitivities are (potentially) desired need to be included. Attach `p` to the user data structure `user_data`. For example, `user_data->p = p;`

    If the user provides a function to evaluate the sensitivity right-hand side, `p` need not be specified.

    •Parameter list (optional)

    If CVODES is to evaluate the right-hand sides of the sensitivity systems, set `plist`, an array of `Ns` integers to specify the parameters `p` with respect to which solution sensitivities are to be computed. If sensitivities with respect to the j-th parameter `p[j]` are desired ($0 \leq$ `j` $<$ `Np`), set $\text{plist}_i = j$, for some $i = 0, \ldots, N_s - 1$.

    If `plist` is not specified, CVODES will compute sensitivities with respect to the first `Ns` parameters; i.e., $\text{plist}_i = i$ ($i = 0, \ldots, N_s - 1$).

    If the user provides a function to evaluate the sensitivity right-hand side, `plist` need not be specified.

    •Parameter scaling factors (optional)

    If CVODES is to estimate tolerances for the sensitivity solution vectors (based on tolerances for the state solution vector) or if CVODES is to evaluate the right-hand sides of the sensitivity systems using the internal difference-quotient function, the results will be more accurate if order of magnitude information is provided.

    Set `pbar`, an array of `Ns` positive scaling factors. Typically, if $p_i \neq 0$, the value $\bar{p}_i = |p_{\text{plist}_i}|$ can be used.

    If `pbar` is not specified, CVODES will use $\bar{p}_i = 1.0$.

    If the user provides a function to evaluate the sensitivity right-hand side and specifies tolerances for the sensitivity variables, `pbar` need not be specified.

    Note that the names for `p`, `pbar`, `plist`, as well as the field p of `user_data` are arbitrary, but they must agree with the arguments passed to `CVodeSetSensParams` below.

14. **Set sensitivity initial conditions**

    Set the `Ns` vectors `yS0[i]` of initial values for sensitivities (for $i =$ 0,..., `Ns` -1), using the appropriate functions defined by the particular NVECTOR implementation chosen.

    First, create an array of `Ns` vectors by making the appropriate call

    ```
    yS0 = N_VCloneVectorArray_***(Ns, y0);
    ```

    or

`yS0 = N_VCloneVectorArrayEmpty_***(Ns, y0);`

Here the argument `y0` serves only to provide the `N_Vector` type for cloning.

Then, for each $i = 0, \ldots,$`Ns` -1, load initial values for the i-th sensitivity vector `yS0[i]`.

15. **Activate sensitivity calculations**

    Call `flag = CVodeSensInit` or `CVodeSensInit1` to activate forward sensitivity computations and allocate internal memory for CVODES related to sensitivity calculations (see §5.2.1).

16. **Set sensitivity tolerances**

    Call `CVodeSensSStolerances`, `CVodeSensSVtolerances` or `CVodeEEtolerances`. (See §5.2.2).

17. **Set sensitivity analysis optional inputs**

    Call `CVodeSetSens*` routines to change from their default values any optional inputs that control the behavior of CVODES in computing forward sensitivities. (See §5.2.5.)

18. **Specify rootfinding**

19. **Advance solution in time**

20. **Extract sensitivity solution**

    After each successful return from `CVode`, the solution of the original IVP is available in the `y` argument of `CVode`, while the sensitivity solution can be extracted into `yS` (which can be the same as `yS0`) by calling one of the routines `CVodeGetSens`,`CVodeGetSens1`, `CVodeGetSensDky`, or `CVodeGetSensDky1` (see §5.2.4).

21. **Get optional outputs**

22. **Deallocate memory for solution vector**

23. **Deallocate memory for sensitivity vectors**

    Upon completion of the integration, deallocate memory for the vectors `yS0` using the appropriate destructor:

    `N_VDestroyVectorArray_***(yS0, Ns);`

    If `yS` was created from `realtype` arrays `yS_i`, it is the user's responsibility to also free the space for the arrays `yS0_i`.

24. **Free user data structure**

25. **Free solver memory**

26. **Free vector specification memory**

27. **Free linear solver and matrix memory**

28. **Finalize MPI, if used**

5.2 User-callable routines for forward sensitivity analysis

This section describes the CVODES functions, in addition to those presented in §4.5, that are called by the user to setup and solve a forward sensitivity problem.

5.2.1 Forward sensitivity initialization and deallocation functions

Activation of forward sensitivity computation is done by calling `CVodeSensInit` or `CVodeSensInit1`, depending on whether the sensitivity right-hand side function returns all sensitivities at once or one by one, respectively. The form of the call to each of these routines is as follows:

`CVodeSensInit`

Call `flag = CVodeSensInit(cvode_mem, Ns, ism, fS, yS0);`

Description The routine `CVodeSensInit` activates forward sensitivity computations and allocates internal memory related to sensitivity calculations.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block returned by `CVodeCreate`.
- `Ns` (`int`) the number of sensitivities to be computed.
- `ism` (`int`) a flag used to select the sensitivity solution method. Its value can be `CV_SIMULTANEOUS` or `CV_STAGGERED`:
  - In the `CV_SIMULTANEOUS` approach, the state and sensitivity variables are corrected at the same time. If `CV_NEWTON` was selected as the nonlinear system solution method, this amounts to performing a modified Newton iteration on the combined nonlinear system;
  - In the `CV_STAGGERED` approach, the correction step for the sensitivity variables takes place at the same time for all sensitivity equations, but only after the correction of the state variables has converged and the state variables have passed the local error test;
- `fS` (`CVSensRhsFn`) is the C function which computes all sensitivity ODE right-hand sides at the same time. For full details see §5.3.
- `yS0` (`N_Vector *`) a pointer to an array of `Ns` vectors containing the initial values of the sensitivities.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeSensInit` was successful.
- `CV_MEM_NULL` The CVODES memory block was not initialized through a previous call to `CVodeCreate`.
- `CV_MEM_FAIL` A memory allocation request has failed.
- `CV_ILL_INPUT` An input argument to `CVodeSensInit` has an illegal value.

Notes Passing `fS=NULL` indicates using the default internal difference quotient sensitivity right-hand side routine.

If an error occurred, `CVodeSensInit` also sends an error message to the error handler function.

It is illegal here to use `ism` = `CV_STAGGERED1`. This option requires a different type for `fS` and can therefore only be used with `CVodeSensInit1` (see below).

`CVodeSensInit1`

Call `flag = CVodeSensInit1(cvode_mem, Ns, ism, fS1, yS0);`

Description The routine `CVodeSensInit1` activates forward sensitivity computations and allocates internal memory related to sensitivity calculations.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block returned by `CVodeCreate`.
- `Ns` (`int`) the number of sensitivities to be computed.
- `ism` (`int`) a flag used to select the sensitivity solution method. Its value can be `CV_SIMULTANEOUS`, `CV_STAGGERED`, or `CV_STAGGERED1`:

- In the CV_SIMULTANEOUS approach, the state and sensitivity variables are corrected at the same time. If CV_NEWTON was selected as the nonlinear system solution method, this amounts to performing a modified Newton iteration on the combined nonlinear system;
- In the CV_STAGGERED approach, the correction step for the sensitivity variables takes place at the same time for all sensitivity equations, but only after the correction of the state variables has converged and the state variables have passed the local error test;
- In the CV_STAGGERED1 approach, all corrections are done sequentially, first for the state variables and then for the sensitivity variables, one parameter at a time. If the sensitivity variables are not included in the error control, this approach is equivalent to CV_STAGGERED. Note that the CV_STAGGERED1 approach can be used only if the user-provided sensitivity right-hand side function is of type CVSensRhs1Fn (see §5.3).

`fS1` (CVSensRhs1Fn) is the C function which computes the right-hand sides of the sensitivity ODE, one at a time. For full details see §5.3.

`yS0` (N_Vector *) a pointer to an array of Ns vectors containing the initial values of the sensitivities.

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

CV_SUCCESS	The call to CVodeSensInit1 was successful.
CV_MEM_NULL	The CVODES memory block was not initialized through a previous call to CVodeCreate.
CV_MEM_FAIL	A memory allocation request has failed.
CV_ILL_INPUT	An input argument to CVodeSensInit1 has an illegal value.

Notes Passing fS1=NULL indicates using the default internal difference quotient sensitivity right-hand side routine.

If an error occurred, CVodeSensInit1 also sends an error message to the error handler funciton.

In terms of the problem size N, number of sensitivity vectors N_s, and maximum method order maxord, the size of the real workspace is increased as follows:

- Base value: lenrw = lenrw + (maxord+5)$N_s N$
- With CVodeSensSVtolerances: lenrw = lenrw +$N_s N$

the size of the integer workspace is increased as follows:

- Base value: leniw = leniw + (maxord+5)$N_s N_i$
- With CVodeSensSVtolerances: leniw = leniw +$N_s N_i$

where N_i is the number of integers in one N_Vector.

The routine CVodeSensReInit, useful during the solution of a sequence of problems of same size, reinitializes the sensitivity-related internal memory. The call to it must follow a call to CVodeSensInit or CVodeSensInit1 (and maybe a call to CVodeReInit). The number Ns of sensitivities is assumed to be unchanged since the call to the initialization function. The call to the CVodeSensReInit function has the form:

CVodeSensReInit

Call `flag = CVodeSensReInit(cvode_mem, ism, yS0);`

Description The routine CVodeSensReInit reinitializes forward sensitivity computations.

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

`ism` (`int`) a flag used to select the sensitivity solution method. Its value can be `CV_SIMULTANEOUS`, `CV_STAGGERED`, or `CV_STAGGERED1`.

`yS0` (`N_Vector *`) a pointer to an array of `Ns` variables of type `N_Vector` containing the initial values of the sensitivities.

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

`CV_SUCCESS` The call to `CVodeReInit` was successful.

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

`CV_NO_SENS` Memory space for sensitivity integration was not allocated through a previous call to `CVodeSensInit`.

`CV_ILL_INPUT` An input argument to `CVodeSensReInit` has an illegal value.

`CV_MEM_FAIL` A memory allocation request has failed.

Notes All arguments of `CVodeSensReInit` are the same as those of the functions `CVodeSensInit` and `CVodeSensInit1`.

If an error occurred, `CVodeSensReInit` also sends a message to the error handler function.

The value of the input argument `ism` must be compatible with the type of the sensitivity ODE right-hand side function. Thus if the sensitivity module was initialized using `CVodeSensInit`, then it is illegal to pass `ism` = `CV_STAGGERED1` to `CVodeSensReInit`.

To deallocate all forward sensitivity-related memory (allocated in a prior call to `CVodeSensInit` or `CVodeSensInit1`), the user must call

`CVodeSensFree`

Call `CVodeSensFree(cvode_mem);`

Description The function `CVodeSensFree` frees the memory allocated for forward sensitivity computations by a previous call to `CVodeSensInit` or `CVodeSensInit1`.

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

Return value The function `CVodeSensFree` has no return value.

Notes In general, `CVodeSensFree` need not be called by the user, as it is invoked automatically by `CVodeFree`.

After a call to `CVodeSensFree`, forward sensitivity computations can be reactivated only by calling `CVodeSensInit` or `CVodeSensInit1` again.

To activate and deactivate forward sensitivity calculations for successive CVODES runs, without having to allocate and deallocate memory, the following function is provided:

`CVodeSensToggleOff`

Call `CVodeSensToggleOff(cvode_mem);`

Description The function `CVodeSensToggleOff` deactivates forward sensitivity calculations. It does *not* deallocate sensitivity-related memory.

Arguments `cvode_mem` (`void *`) pointer to the memory previously returned by `CVodeCreate`.

Return value The return value `flag` of `CVodeSensToggle` is one of:

`CV_SUCCESS` `CVodeSensToggleOff` was successful.

`CV_MEM_NULL` `cvode_mem` was `NULL`.

Notes Since sensitivity-related memory is not deallocated, sensitivities can be reactivated at a later time (using `CVodeSensReInit`).

5.2.2 Forward sensitivity tolerance specification functions

One of the following three functions must be called to specify the integration tolerances for sensitivities. Note that this call must be made after the call to `CVodeSensInit`/`CVodeSensInit1`.

`CVodeSensSStolerances`

Call `flag = CVodeSensSStolerances(cvode_mem, reltolS, abstolS);`

Description The function `CVodeSensSStolerances` specifies scalar relative and absolute tolerances.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block returned by `CVodeCreate`.
- `reltolS` (`realtype`) is the scalar relative error tolerance.
- `abstolS` (`realtype*`) is a pointer to an array of length `Ns` containing the scalar absolute error tolerances, one for each parameter.

Return value The return flag `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeSStolerances` was successful.
- `CV_MEM_NULL` The CVODES memory block was not initialized through a previous call to `CVodeCreate`.
- `CV_NO_SENS` The sensitivity allocation function (`CVodeSensInit` or `CVodeSensInit1`) has not been called.
- `CV_ILL_INPUT` One of the input tolerances was negative.

`CVodeSensSVtolerances`

Call `flag = CVodeSensSVtolerances(cvode_mem, reltolS, abstolS);`

Description The function `CVodeSensSVtolerances` specifies scalar relative tolerance and vector absolute tolerances.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block returned by `CVodeCreate`.
- `reltolS` (`realtype`) is the scalar relative error tolerance.
- `abstolS` (`N_Vector*`) is an array of `Ns` variables of type `N_Vector`. The `N_Vector` from `abstolS[is]` specifies the vector tolerances for `is`-th sensitivity.

Return value The return flag `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeSVtolerances` was successful.
- `CV_MEM_NULL` The CVODES memory block was not initialized through a previous call to `CVodeCreate`.
- `CV_NO_SENS` The allocation function for sensitivities has not been called.
- `CV_ILL_INPUT` The relative error tolerance was negative or an absolute tolerance vector had a negative component.

Notes This choice of tolerances is important when the absolute error tolerance needs to be different for each component of any vector `yS[i]`.

`CVodeSensEEtolerances`

Call `flag = CVodeSensEEtolerances(cvode_mem);`

Description When `CVodeSensEEtolerances` is called, CVODES will estimate tolerances for sensitivity variables based on the tolerances supplied for states variables and the scaling factors $\bar{p}$.

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

Return value The return flag `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeSensEEtolerances` was successful.
- `CV_MEM_NULL` The CVODES memory block was not initialized through a previous call to `CVodeCreate`.
- `CV_NO_SENS` The sensitivity allocation function has not been called.

5.2.3 CVODES solver function

Even if forward sensitivity analysis was enabled, the call to the main solver function `CVode` is exactly the same as in §4.5.5. However, in this case the return value `flag` can also be one of the following:

`CV_SRHSFUNC_FAIL`	The sensitivity right-hand side function failed in an unrecoverable manner.
`CV_FIRST_SRHSFUNC_ERR`	The sensitivity right-hand side function failed at the first call.
`CV_REPTD_SRHSFUNC_ERR`	Convergence tests occurred too many times due to repeated recoverable errors in the sensitivity right-hand side function. This flag will also be returned if the sensitivity right-hand side function had repeated recoverable errors during the estimation of an initial step size.
`CV_UNREC_SRHSFUNC_ERR`	The sensitivity right-hand function had a recoverable error, but no recovery was possible. This failure mode is rare, as it can occur only if the sensitivity right-hand side function fails recoverably after an error test failed while at order one.

5.2.4 Forward sensitivity extraction functions

If forward sensitivity computations have been initialized by a call to `CVodeSensInit`/`CVodeSensInit1`, or reinitialized by a call to `CVSensReInit`, then CVODES computes both a solution and sensitivities at time `t`. However, `CVode` will still return only the solution y in `yout`. Solution sensitivities can be obtained through one of the following functions:

`CVodeGetSens`

Call `flag = CVodeGetSens(cvode_mem, &tret, yS);`

Description The function `CVodeGetSens` returns the sensitivity solution vectors after a successful return from `CVode`.

Arguments
- `cvode_mem` (`void *`) pointer to the memory previously allocated by `CVodeInit`.
- `tret` (`realtype *`) the time reached by the solver (output).
- `yS` (`N_Vector *`) array of computed forward sensitivity vectors.

Return value The return value `flag` of `CVodeGetSens` is one of:
- `CV_SUCCESS` `CVodeGetSens` was successful.
- `CV_MEM_NULL` `cvode_mem` was `NULL`.
- `CV_NO_SENS` Forward sensitivity analysis was not initialized.
- `CV_BAD_DKY` `yS` is `NULL`.

Notes Note that the argument `tret` is an output for this function. Its value will be the same as that returned at the last `CVode` call.

The function `CVodeGetSensDky` computes the `k`-th derivatives of the interpolating polynomials for the sensitivity variables at time `t`. This function is called by `CVodeGetSens` with `k` $= 0$, but may also be called directly by the user.

`CVodeGetSensDky`

Call `flag = CVodeGetSensDky(cvode_mem, t, k, dkyS);`

Description The function `CVodeGetSensDky` returns derivatives of the sensitivity solution vectors after a successful return from `CVode`.

Arguments
- `cvode_mem` (`void *`) pointer to the memory previously allocated by `CVodeInit`.
- `t` (`realtype`) specifies the time at which sensitivity information is requested. The time `t` must fall within the interval defined by the last successful step taken by CVODES.
- `k` (`int`) order of derivatives.

dkyS (N_Vector *) array of Ns vectors containing the derivatives on output. The space for dkyS must be allocated by the user.

Return value The return value flag of CVodeGetSensDky is one of:

CV_SUCCESS CVodeGetSensDky succeeded.
CV_MEM_NULL cvode_mem was NULL.
CV_NO_SENS Forward sensitivity analysis was not initialized.
CV_BAD_DKY One of the vectors dkyS is NULL.
CV_BAD_K k is not in the range $0, 1, ...,$ qlast.
CV_BAD_T The time t is not in the allowed range.

Forward sensitivity solution vectors can also be extracted separately for each parameter in turn through the functions CVodeGetSens1 and CVodeGetSensDky1, defined as follows:

CVodeGetSens1

Call flag = CVodeGetSens1(cvode_mem, &tret, is, yS);

Description The function CVodeGetSens1 returns the is-th sensitivity solution vector after a successful return from CVode.

Arguments
cvode_mem (void *) pointer to the memory previously allocated by CVodeInit.
tret (realtype *) the time reached by the solver (output).
is (int) specifies which sensitivity vector is to be returned ($0 \leq$is$< N_s$).
yS (N_Vector) the computed forward sensitivity vector.

Return value The return value flag of CVodeGetSens1 is one of:

CV_SUCCESS CVodeGetSens1 was successful.
CV_MEM_NULL cvode_mem was NULL.
CV_NO_SENS Forward sensitivity analysis was not initialized.
CV_BAD_IS The index is is not in the allowed range.
CV_BAD_DKY yS is NULL.
CV_BAD_T The time t is not in the allowed range.

Notes Note that the argument tret is an output for this function. Its value will be the same as that returned at the last CVode call.

CVodeGetSensDky1

Call flag = CVodeGetSensDky1(cvode_mem, t, k, is, dkyS);

Description The function CVodeGetSensDky1 returns the k-th derivative of the is-th sensitivity solution vector after a successful return from CVode.

Arguments
cvode_mem (void *) pointer to the memory previously allocated by CVodeInit.
t (realtype) specifies the time at which sensitivity information is requested. The time t must fall within the interval defined by the last successful step taken by CVODES.
k (int) order of derivative.
is (int) specifies the sensitivity derivative vector to be returned ($0 \leq$is$< N_s$).
dkyS (N_Vector) the vector containing the derivative. The space for dkyS must be allocated by the user.

Return value The return value flag of CVodeGetSensDky1 is one of:

CV_SUCCESS CVodeGetQuadDky1 succeeded.
CV_MEM_NULL The pointer to cvode_mem was NULL.
CV_NO_SENS Forward sensitivity analysis was not initialized.

CV_BAD_DKY	dkyS or one of the vectors dkyS[i] is NULL.
CV_BAD_IS	The index is is not in the allowed range.
CV_BAD_K	k is not in the range $0, 1, ...,$ qlast.
CV_BAD_T	The time t is not in the allowed range.

5.2.5 Optional inputs for forward sensitivity analysis

Optional input variables that control the computation of sensitivities can be changed from their default values through calls to CVodeSetSens* functions. Table 5.1 lists all forward sensitivity optional input functions in CVODES which are described in detail in the remainder of this section.

CVodeSetSensParams

Call `flag = CVodeSetSensParams(cvode_mem, p, pbar, plist);`

Description The function CVodeSetSensParams specifies problem parameter information for sensitivity calculations.

Arguments
- `cvode_mem` (void *) pointer to the CVODES memory block.
- `p` (realtype *) a pointer to the array of real problem parameters used to evaluate $f(t, y, p)$. If non-NULL, p must point to a field in the user's data structure user_data passed to the right-hand side function. (See §5.1).
- `pbar` (realtype *) an array of Ns positive scaling factors. If non-NULL, pbar must have all its components > 0.0. (See §5.1).
- `plist` (int *) an array of Ns non-negative indices to specify which components p[i] to use in estimating the sensitivity equations. If non-NULL, plist must have all components ≥ 0. (See §5.1).

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_NO_SENS	Forward sensitivity analysis was not initialized.
CV_ILL_INPUT	An argument has an illegal value.

Notes This function must be preceded by a call to CVodeSensInit or CVodeSensInit1.

CVodeSetSensDQMethod

Call `flag = CVodeSetSensDQMethod(cvode_mem, DQtype, DQrhomax);`

Description The function CVodeSetSensDQMethod specifies the difference quotient strategy in the case in which the right-hand side of the sensitivity equations are to be computed by CVODES.

Arguments
- `cvode_mem` (void *) pointer to the CVODES memory block.
- `DQtype` (int) specifies the difference quotient type. Its value can be CV_CENTERED or CV_FORWARD.

Table 5.1: Forward sensitivity optional inputs

Optional input	Routine name	Default
Sensitivity scaling factors	CVodeSetSensParams	NULL
DQ approximation method	CVodeSetSensDQMethod	centered/0.0
Error control strategy	CVodeSetSensErrCon	SUNFALSE
Maximum no. of nonlinear iterations	CVodeSetSensMaxNonlinIters	3

DQrhomax (realtype) positive value of the selection parameter used in deciding switching between a simultaneous or separate approximation of the two terms in the sensitivity right-hand side.

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 An argument has an illegal value.

Notes If DQrhomax = 0.0, then no switching is performed. The approximation is done simultaneously using either centered or forward finite differences, depending on the value of DQtype. For values of DQrhomax $\geq$ 1.0, the simultaneous approximation is used whenever the estimated finite difference perturbations for states and parameters are within a factor of DQrhomax, and the separate approximation is used otherwise. Note that a value DQrhomax $<$ 1.0 will effectively disable switching. See §2.6 for more details.

The default value are DQtype=CV_CENTERED and DQrhomax= 0.0.

CVodeSetSensErrCon

Call flag = CVodeSetSensErrCon(cvode_mem, errconS);

Description The function CVodeSetSensErrCon specifies the error control strategy for sensitivity variables.

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

errconS (booleantype) specifies whether sensitivity variables are to be included (SUNTRUE) or not (SUNFALSE) in the error control mechanism.

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes By default, errconS is set to SUNFALSE. If errconS=SUNTRUE then both state variables and sensitivity variables are included in the error tests. If errconS=SUNFALSE then the sensitivity variables are excluded from the error tests. Note that, in any event, all variables are considered in the convergence tests.

CVodeSetSensMaxNonlinIters

Call flag = CVodeSetSensMaxNonlinIters(cvode_mem, maxcorS);

Description The function CVodeSetSensMaxNonlinIters specifies the maximum number of nonlinear solver iterations for sensitivity variables per step.

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

maxcorS (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.

Notes The default value is 3.

5.2.6 Optional outputs for forward sensitivity analysis

Optional output functions that return statistics and solver performance information related to forward sensitivity computations are listed in Table 5.2 and described in detail in the remainder of this section.

`CVodeGetSensNumRhsEvals`

Call `flag = CVodeGetSensNumRhsEvals(cvode_mem, &nfSevals);`

Description The function `CVodeGetSensNumRhsEvals` returns the number of calls to the sensitivity right-hand side function.

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

`nfSevals` (`long int`) number of calls to the sensitivity right-hand side function.

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

`CV_SUCCESS` The optional output value has been successfully set.

`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes In order to accommodate any of the three possible sensitivity solution methods, the default internal finite difference quotient functions evaluate the sensitivity right-hand sides one at a time. Therefore, `nfSevals` will always be a multiple of the number of sensitivity parameters (the same as the case in which the user supplies a routine of type `CVSensRhs1Fn`).

`CVodeGetNumRhsEvalsSens`

Call `flag = CVodeGetNumRhsEvalsSens(cvode_mem, &nfevalsS);`

Description The function `CVodeGetNumRhsEvalsSEns` returns the number of calls to the user's right-hand side function due to the internal finite difference approximation of the sensitivity right-hand sides.

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

`nfevalsS` (`long int`) number of calls to the user's ODE right-hand side function for the evaluation of sensitivity right-hand sides.

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

`CV_SUCCESS` The optional output value has been successfully set.

`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes This counter is incremented only if the internal finite difference approximation routines are used for the evaluation of the sensitivity right-hand sides.

`CVodeGetSensNumErrTestFails`

Call `flag = CVodeGetSensNumErrTestFails(cvode_mem, &nSetfails);`

Table 5.2: Forward sensitivity optional outputs

Optional output	**Routine name**
No. of calls to sensitivity r.h.s. function	`CVodeGetSensNumRhsEvals`
No. of calls to r.h.s. function for sensitivity	`CVodeGetNumRhsEvalsSens`
No. of sensitivity local error test failures	`CVodeGetSensNumErrTestFails`
No. of calls to lin. solv. setup routine for sens.	`CVodeGetSensNumLinSolvSetups`
Error weight vector for sensitivity variables	`CVodeGetSensErrWeights`
No. of sens. nonlinear solver iterations	`CVodeGetSensNumNonlinSolvIters`
No. of sens. convergence failures	`CVodeGetSensNumNonlinSolvConvFails`
No. of staggered nonlinear solver iterations	`CVodeGetStgrSensNumNonlinSolvIters`
No. of staggered convergence failures	`CVodeGetStgrSensNumNonlinSolvConvFails`

Description The function `CVodeGetSensNumErrTestFails` returns the number of local error test failures for the sensitivity variables that have occurred.

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

`nSetfails` (`long int`) number of error test failures.

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

`CV_SUCCESS` The optional output value has been successfully set.

`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes This counter is incremented only if the sensitivity variables have been included in the error test (see `CVodeSetSensErrCon` in §5.2.5). Even in that case, this counter is not incremented if the `ism`=`CV_SIMULTANEOUS` sensitivity solution method has been used.

`CVodeGetSensNumLinSolvSetups`

Call `flag = CVodeGetSensNumLinSolvSetups(cvode_mem, &nlinsetupsS);`

Description The function `CVodeGetSensNumLinSolvSetups` returns the number of calls to the linear solver setup function due to forward sensitivity calculations.

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

`nlinsetupsS` (`long int`) number of calls 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`.

`CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes This counter is incremented only if Newton iteration has been used and if either the `ism` = `CV_STAGGERED` or the `ism` = `CV_STAGGERED1` sensitivity solution method has been specified (see §5.2.1).

`CVodeGetSensStats`

Call
```
flag = CVodeGetSensStats(cvode_mem, &nfSevals, &nfevalsS, &nSetfails,
                         &nSetfails, &nlinsetupsS);
```

Description The function `CVodeGetSensStats` returns all of the above sensitivity-related solver statistics as a group.

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

`nfSevals` (`long int`) number of calls to the sensitivity right-hand side function.

`nfevalsS` (`long int`) number of calls to the ODE right-hand side function for sensitivity evaluations.

`nSetfails` (`long int`) number of error test failures.

`nlinsetupsS` (`long int`) number of calls to the linear solver setup function.

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_NO_SENS` Forward sensitivity analysis was not initialized.

`CVodeGetSensErrWeights`

Call `flag = CVodeGetSensErrWeights(cvode_mem, eSweight);`

Description The function `CVodeGetSensErrWeights` returns the sensitivity error weight vectors at the current time. These are the reciprocals of the W_i of (2.7) for the sensitivity variables.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`eSweight` (`N_Vector *`) pointer to the array of error weight vectors.

Return value The return value `flag` (of type `int`) is one of:
`CV_SUCCESS` The optional output value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes The user must allocate memory for `eweightS`.

`CVodeGetSensNumNonlinSolvIters`

Call `flag = CVodeGetSensNumNonlinSolvIters(cvode_mem, &nSniters);`

Description The function `CVodeGetSensNumNonlinSolvIters` returns the number of nonlinear iterations performed for sensitivity calculations.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`nSniters` (`long int`) number of nonlinear iterations performed.

Return value The return value `flag` (of type `int`) is one of:
`CV_SUCCESS` The optional output value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes This counter is incremented only if `ism` was `CV_STAGGERED` or `CV_STAGGERED1` (see §5.2.1).

In the `CV_STAGGERED1` case, the value of `nSniters` is the sum of the number of nonlinear iterations performed for each sensitivity equation. These individual counters can be obtained through a call to `CVodeGetStgrSensNumNonlinSolvIters` (see below).

`CVodeGetSensNumNonlinSolvConvFails`

Call `flag = CVodeGetSensNumNonlinSolvConvFails(cvode_mem, &nSncfails);`

Description The function `CVodeGetSensNumNonlinSolvConvFails` returns the number of nonlinear convergence failures that have occurred for sensitivity calculations.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`nSncfails` (`long int`) number of nonlinear convergence failures.

Return value The return value `flag` (of type `int`) is one of:
`CV_SUCCESS` The optional output value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes This counter is incremented only if `ism` was `CV_STAGGERED` or `CV_STAGGERED1` (see §5.2.1).

In the `CV_STAGGERED1` case, the value of `nSncfails` is the sum of the number of nonlinear convergence failures that occurred for each sensitivity equation. These individual counters can be obtained through a call to `CVodeGetStgrSensNumNonlinConvFails` (see below).

`CVodeGetSensNonlinSolvStats`

Call `flag = CVodeGetSensNonlinSolvStats(cvode_mem, &nSniters, &nSncfails);`

Description The function `CVodeGetSensNonlinSolvStats` returns the sensitivity-related nonlinear solver statistics as a group.

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

`nSniters` (`long int`) number of nonlinear iterations performed.

`nSncfails` (`long int`) number of nonlinear convergence failures.

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_NO_SENS` Forward sensitivity analysis was not initialized.

`CVodeGetStgrSensNumNonlinSolvIters`

Call `flag = CVodeGetStgrSensNumNonlinSolvIters(cvode_mem, nSTGR1niters);`

Description The function `CVodeGetStgrSensNumNonlinSolvIters` returns the number of nonlinear (functional or Newton) iterations performed for each sensitivity equation separately, in the `CV_STAGGERED1` case.

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

`nSTGR1niters` (`long int *`) an array (of dimension `Ns`) which will be set with the number of nonlinear iterations performed for each sensitivity system individually.

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

`CV_SUCCESS` The optional output value has been successfully set.

`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes The user must allocate space for `nSTGR1niters`.

`CVodeGetStgrSensNumNonlinSolvConvFails`

Call `flag = CVodeGetStgrSensNumNonlinSolvConvFails(cvode_mem, nSTGR1ncfails);`

Description The function `CVodeGetStgrSensNumNonlinSolvConvFails` returns the number of nonlinear convergence failures that have occurred for each sensitivity equation separately, in the `CV_STAGGERED1` case.

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

`nSTGR1ncfails` (`long int *`) an array (of dimension `Ns`) which will be set with the number of nonlinear convergence failures for each sensitivity system individually.

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

`CV_SUCCESS` The optional output value has been successfully set.

`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes The user must allocate space for `nSTGR1ncfails`.

5.3 User-supplied routines for forward sensitivity analysis

In addition to the required and optional user-supplied routines described in §4.6, when using CVODES for forward sensitivity analysis, the user has the option of providing a routine that calculates the right-hand side of the sensitivity equations (2.11).

By default, CVODES uses difference quotient approximation routines for the right-hand sides of the sensitivity equations. However, CVODES allows the option for user-defined sensitivity right-hand side routines (which also provides a mechanism for interfacing CVODES to routines generated by automatic differentiation).

5.3.1 Sensitivity equations right-hand side (all at once)

If the `CV_SIMULTANEOUS` or `CV_STAGGERED` approach was selected in the call to `CVodeSensInit` or `CVodeSensInit1`, the user may provide the right-hand sides of the sensitivity equations (2.11), for all sensitivity parameters at once, through a function of type `CVSensRhsFn` defined by:

`CVSensRhsFn`

Definition
```
typedef int (*CVSensRhsFn)(int Ns, realtype t,
                           N_Vector y, N_Vector ydot,
                           N_Vector *yS, N_Vector *ySdot,
                           void *user_data,
                           N_Vector tmp1, N_Vector tmp2);
```

Purpose This function computes the sensitivity right-hand side for all sensitivity equations at once. It must compute the vectors $(\partial f/\partial y)s_i(t)+(\partial f/\partial p_i)$ and store them in `ySdot[i]`.

Arguments
- `t` is the current value of the independent variable.
- `y` is the current value of the state vector, $y(t)$.
- `ydot` is the current value of the right-hand side of the state equations.
- `yS` contains the current values of the sensitivity vectors.
- `ySdot` is the output of `CVSensRhsFn`. On exit it must contain the sensitivity right-hand side vectors.
- `user_data` is a pointer to user data, the same as the `user_data` parameter passed to `CVodeSetUserData`.
- `tmp1`
- `tmp2` are `N_Vectors` of length N which can be used as temporary storage.

Return value A `CVSensRhsFn` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CV_SRHSFUNC_FAIL` is returned).

Notes A sensitivity right-hand side function of type `CVSensRhsFn` is not compatible with the `CV_STAGGERED1` approach.

Allocation of memory for `ySdot` is handled within CVODES.

There are two situations in which recovery is not possible even if `CVSensRhsFn` function returns a recoverable error flag. One is when this occurs at the very first call to the `CVSensRhsFn` (in which case CVODES returns `CV_FIRST_SRHSFUNC_ERR`). The other is when a recoverable error is reported by `CVSensRhsFn` after an error test failure, while the linear multistep method order is equal to 1 (in which case CVODES returns `CV_UNREC_SRHSFUNC_ERR`).

5.3.2 Sensitivity equations right-hand side (one at a time)

Alternatively, the user may provide the sensitivity right-hand sides, one sensitivity parameter at a time, through a function of type CVSensRhs1Fn. Note that a sensitivity right-hand side function of type CVSensRhs1Fn is compatible with any valid value of the argument ism to CVodeSensInit and CVodeSensInit1, and is *required* if ism = CV_STAGGERED1 in the call to CVodeSensInit1. The type CVSensRhs1Fn is defined by

CVSensRhs1Fn

Definition
```
typedef int (*CVSensRhs1Fn)(int Ns, realtype t,
                            N_Vector y, N_Vector ydot,
                            int iS, N_Vector yS, N_Vector ySdot,
                            void *user_data,
                            N_Vector tmp1, N_Vector tmp2);
```

Purpose This function computes the sensitivity right-hand side for one sensitivity equation at a time. It must compute the vector $(\partial f/\partial y)s_i(t) + (\partial f/\partial p_i)$ for $i =$ iS and store it in ySdot.

Arguments
- t is the current value of the independent variable.
- y is the current value of the state vector, $y(t)$.
- ydot is the current value of the right-hand side of the state equations.
- iS is the index of the parameter for which the sensitivity right-hand side must be computed ($0 \leq$ iS $<$ Ns).
- yS contains the current value of the iS-th sensitivity vector.
- ySdot is the output of CVSensRhs1Fn. On exit it must contain the iS-th sensitivity right-hand side vector.
- user_data is a pointer to user data, the same as the user_data parameter passed to CVodeSetUserData.
- tmp1
- tmp2 are N_Vectors of length N which can be used as temporary storage.

Return value A CVSensRhs1Fn should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and CV_SRHSFUNC_FAIL is returned).

Notes Allocation of memory for ySdot is handled within CVODES.

There are two situations in which recovery is not possible even if CVSensRhs1Fn function returns a recoverable error flag. One is when this occurs at the very first call to the CVSensRhs1Fn (in which case CVODES returns CV_FIRST_SRHSFUNC_ERR). The other is when a recoverable error is reported by CVSensRhs1Fn after an error test failure, while the linear multistep method order equal to 1 (in which case CVODES returns CV_UNREC_SRHSFUNC_ERR).

5.4 Integration of quadrature equations depending on forward sensitivities

CVODES provides support for integration of quadrature equations that depends not only on the state variables but also on forward sensitivities.

The following is an overview of the sequence of calls in a user's main program in this situation. Steps that are unchanged from the skeleton program presented in §5.1 are grayed out.

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

2. **Set problem dimensions etc.**
3. **Set vectors of initial values**
4. **Create CVODES object**
5. **Initialize CVODES**
6. **Specify integration tolerances**
7. **Set optional inputs**
8. **Create matrix object**
9. **Create linear solver object**
10. **Set linear solver optional inputs**
11. **Initialize sensitivity-independent quadrature problem**
12. **Define the sensitivity problem**
13. **Set sensitivity initial conditions**
14. **Activate sensitivity calculations**
15. **Set sensitivity analysis optional inputs**
16. **Set vector of initial values for quadrature variables**

    Typically, the quadrature variables should be initialized to 0.
17. **Initialize sensitivity-dependent quadrature integration**

    Call `CVodeQuadSensInit` to specify the quadrature equation right-hand side function and to allocate internal memory related to quadrature integration. See §5.4.1 for details.
18. **Set optional inputs for sensitivity-dependent quadrature integration**

    Call `CVodeSetQuadSensErrCon` to indicate whether or not quadrature variables should be used in the step size control mechanism. If so, one of the `CVodeQuadSens*tolerances` functions must be called to specify the integration tolerances for quadrature variables. See §5.4.4 for details.
19. **Advance solution in time**
20. **Extract sensitivity-dependent quadrature variables**

    Call `CVodeGetQuadSens`, `CVodeGetQuadSens1`, `CVodeGetQuadSensDky` or `CVodeGetQuadSensDky1` to obtain the values of the quadrature variables or their derivatives at the current time. See §5.4.3 for details.
21. **Get optional outputs**
22. **Extract sensitivity solution**
23. **Get sensitivity-dependent quadrature optional outputs**

    Call `CVodeGetQuadSens*` functions to obtain desired optional output related to the integration of sensitivity-dependent quadratures. See §5.4.5 for details.
24. **Deallocate memory for solutions vector**
25. **Deallocate memory for sensitivity vectors**

26. **Deallocate memory for sensitivity-dependent quadrature variables**

27. **Free vector specification memory**

28. **Free linear solver and matrix memory**

29. **Free solver memory**

30. **Finalize MPI, if used**

Note: `CVodeQuadSensInit` (step 17 above) can be called and quadrature-related optional inputs (step 18 above) can be set anywhere between steps 12 and 19.

5.4.1 Sensitivity-dependent quadrature initialization and deallocation

The function `CVodeQuadSensInit` activates integration of quadrature equations depending on sensitivities and allocates internal memory related to these calculations. If `rhsQS` is input as `NULL`, then CVODES uses an internal function that computes difference quotient approximations to the functions $\bar{q}_i = q_y s_i + q_{p_i}$, in the notation of (2.9). The form of the call to this function is as follows:

`CVodeQuadSensInit`

Call `flag = CVodeQuadSensInit(cvode_mem, rhsQS, yQS0);`

Description The function `CVodeQuadSensInit` provides required problem specifications, allocates internal memory, and initializes quadrature integration.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block returned by `CVodeCreate`.
- `rhsQS` (`CVQuadSensRhsFn`) is the C function which computes f_{QS}, the right-hand side of the sensitivity-dependent quadrature equations (for full details see §5.4.6).
- `yQS0` (`N_Vector *`) contains the initial values of sensitivity-dependent quadratures.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeQuadSensInit` was successful.
- `CVODE_MEM_NULL` The CVODES memory was not initialized by a prior call to `CVodeCreate`.
- `CVODE_MEM_FAIL` A memory allocation request failed.
- `CV_NO_SENS` The sensitivities were not initialized by a prior call to `CVodeSensInit` or `CVodeSensInit1`.
- `CV_ILL_INPUT` The parameter `yQS0` is `NULL`.

Notes Before calling `CVodeQuadSensInit`, the user must enable the sensitivites by calling `CVodeSensInit` or `CVodeSensInit1`.

If an error occurred, `CVodeQuadSensInit` also sends an error message to the error handler function.

In terms of the number of quadrature variables N_q and maximum method order `maxord`, the size of the real workspace is increased as follows:

- Base value: `lenrw` = `lenrw` + (`maxord`+5)N_q
- If `CVodeQuadSensSVtolerances` is called: `lenrw` = `lenrw` +$N_q N_s$

and the size of the integer workspace is increased as follows:

- Base value: `leniw` = `leniw` + (`maxord`+5)N_q
- If `CVodeQuadSensSVtolerances` is called: `leniw` = `leniw` +$N_q N_s$

The function CVodeQuadSensReInit, useful during the solution of a sequence of problems of same size, reinitializes quadrature-related internal memory and must follow a call to CVodeQuadSensInit. The number Nq of quadratures as well as the number Ns of sensitivities are assumed to be unchanged from the prior call to CVodeQuadSensInit. The call to the CVodeQuadSensReInit function has the form:

CVodeQuadSensReInit

Call `flag = CVodeQuadSensReInit(cvode_mem, yQS0);`

Description The function CVodeQuadSensReInit provides required problem specifications and reinitializes the sensitivity-dependent quadrature integration.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `yQS0` (`N_Vector *`) contains the initial values of sensitivity-dependent quadratures.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to CVodeQuadSensReInit was successful.
- `CVODE_MEM_NULL` The CVODES memory was not initialized by a prior call to CVodeCreate.
- `CV_NO_SENS` Memory space for the sensitivity calculation was not allocated by a prior call to CVodeSensInit or CVodeSensInit1.
- `CV_NO_QUADSENS` Memory space for the sensitivity quadratures integration was not allocated by a prior call to CVodeQuadSensInit.
- `CV_ILL_INPUT` The parameter yQS0 is NULL.

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

CVodeQuadSensFree

Call `CVodeQuadSensFree(cvode_mem);`

Description The function CVodeQuadSensFree frees the memory allocated for sensitivity quadrature integration.

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

Return value The function CVodeQuadSensFree has no return value.

Notes In general, CVodeQuadSensFree need not be called by the user, as it is invoked automatically by CVodeFree.

5.4.2 CVODES solver function

Even if quadrature integration was enabled, the call to the main solver function CVode is exactly the same as in §4.5.5. However, in this case the return value `flag` can also be one of the following:

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

5.4.3 Sensitivity-dependent quadrature extraction functions

If sensitivity-dependent quadratures have been initialized by a call to `CVodeQuadSensInit`, or reinitialized by a call to `CVodeQuadSensReInit`, then CVODES computes a solution, sensitivity vectors, and quadratures depending on sensitivities at time `t`. However, `CVode` will still return only the solution y. Sensitivity-dependent quadratures can be obtained using one of the following functions:

`CVodeGetQuadSens`

Call `flag = CVodeGetQuadSens(cvode_mem, &tret, yQS);`

Description The function `CVodeGetQuadSens` returns the quadrature sensitivities solution vectors after a successful return from `CVode`.

Arguments
- `cvode_mem` (`void *`) pointer to the memory previously allocated by `CVodeInit`.
- `tret` (`realtype`) the time reached by the solver (output).
- `yQS` (`N_Vector *`) array of `Ns` computed sensitivity-dependent quadrature vectors.

Return value The return value `flag` of `CVodeGetQuadSens` is one of:

`CV_SUCCESS`	`CVodeGetQuadSens` was successful.
`CVODE_MEM_NULL`	`cvode_mem` was NULL.
`CV_NO_SENS`	Sensitivities were not activated.
`CV_NO_QUADSENS`	Quadratures depending on the sensitivities were not activated.
`CV_BAD_DKY`	`yQS` or one of the `yQS[i]` is `NULL`.

The function `CVodeGetQuadSensDky` computes the `k`-th derivatives of the interpolating polynomials for the sensitivity-dependent quadrature variables at time `t`. This function is called by `CVodeGetQuadSens` with `k = 0`, but may also be called directly by the user.

`CVodeGetQuadSensDky`

Call `flag = CVodeGetQuadSensDky(cvode_mem, t, k, dkyQS);`

Description The function `CVodeGetQuadSensDky` returns derivatives of the quadrature sensitivities solution vectors after a successful return from `CVode`.

Arguments
- `cvode_mem` (`void *`) pointer to the memory previously allocated by `CVodeInit`.
- `t` (`realtype`) the time at which information is requested. The time `t` must fall within the interval defined by the last successful step taken by CVODES.
- `k` (`int`) order of the requested derivative.
- `dkyQS` (`N_Vector *`) array of `Ns` the vector containing the derivatives on output. This vector array must be allocated by the user.

Return value The return value `flag` of `CVodeGetQuadSensDky` is one of:

`CV_SUCCESS`	`CVodeGetQuadSensDky` succeeded.
`CVODE_MEM_NULL`	The pointer to `cvode_mem` was `NULL`.
`CV_NO_SENS`	Sensitivities were not activated.
`CV_NO_QUADSENS`	Quadratures depending on the sensitivities were not activated.
`CV_BAD_DKY`	`dkyQS` or one of the vectors `dkyQS[i]` is `NULL`.
`CV_BAD_K`	`k` is not in the range $0, 1, ...,$ `qlast`.
`CV_BAD_T`	The time `t` is not in the allowed range.

Quadrature sensitivity solution vectors can also be extracted separately for each parameter in turn through the functions `CVodeGetQuadSens1` and `CVodeGetQuadSensDky1`, defined as follows:

`CVodeGetQuadSens1`

Call `flag = CVodeGetQuadSens1(cvode_mem, &tret, is, yQS);`

Description The function `CVodeGetQuadSens1` returns the `is`-th sensitivity of quadratures after a successful return from `CVode`.

Arguments
- `cvode_mem` (`void *`) pointer to the memory previously allocated by `CVodeInit`.
- `tret` (`realtype`) the time reached by the solver (output).
- `is` (`int`) specifies which sensitivity vector is to be returned ($0 \leq$ `is` $< N_s$).
- `yQS` (`N_Vector`) the computed sensitivity-dependent quadrature vector.

Return value The return value `flag` of `CVodeGetQuadSens1` is one of:

`CV_SUCCESS`	`CVodeGetQuadSens1` was successful.
`CVODE_MEM_NULL`	`cvode_mem` was `NULL`.
`CV_NO_SENS`	Forward sensitivity analysis was not initialized.
`CV_NO_QUADSENS`	Quadratures depending on the sensitivities were not activated.
`CV_BAD_IS`	The index `is` is not in the allowed range.
`CV_BAD_DKY`	`yQS` is `NULL`.

`CVodeGetQuadSensDky1`

Call `flag = CVodeGetQuadSensDky1(cvode_mem, t, k, is, dkyQS);`

Description The function `CVodeGetQuadSensDky1` returns the `k`-th derivative of the `is`-th sensitivity solution vector after a successful return from `CVode`.

Arguments
- `cvode_mem` (`void *`) pointer to the memory previously allocated by `CVodeInit`.
- `t` (`realtype`) specifies the time at which sensitivity information is requested. The time `t` must fall within the interval defined by the last successful step taken by CVODES.
- `k` (`int`) order of derivative.
- `is` (`int`) specifies the sensitivity derivative vector to be returned ($0 \leq$`is`$< N_s$).
- `dkyQS` (`N_Vector`) the vector containing the derivative on output. The space for `dkyQS` must be allocated by the user.

Return value The return value `flag` of `CVodeGetQuadSensDky1` is one of:

`CV_SUCCESS`	`CVodeGetQuadDky1` succeeded.
`CVODE_MEM_NULL`	`cvode_mem` was `NULL`.
`CV_NO_SENS`	Forward sensitivity analysis was not initialized.
`CV_NO_QUADSENS`	Quadratures depending on the sensitivities were not activated.
`CV_BAD_DKY`	`dkyQS` is `NULL`.
`CV_BAD_IS`	The index `is` is not in the allowed range.
`CV_BAD_K`	`k` is not in the range $0, 1, ...,$ `qlast`.
`CV_BAD_T`	The time `t` is not in the allowed range.

5.4.4 Optional inputs for sensitivity-dependent quadrature integration

CVODES provides the following optional input functions to control the integration of sensitivity-dependent quadrature equations.

`CVodeSetQuadSensErrCon`

Call `flag = CVodeSetQuadSensErrCon(cvode_mem, errconQS)`

Description The function `CVodeSetQuadSensErrCon` specifies whether or not the quadrature variables are to be used in the step size control mechanism. If they are, the user must call one of the functions `CVodeQuadSensSStolerances`, `CVodeQuadSensSVtolerances`, or `CVodeQuadSensEEtolerances` to specify the integration tolerances for the quadrature variables.

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

`errconQS` (`booleantype`) specifies whether sensitivity quadrature variables are to be included (`SUNTRUE`) or not (`SUNFALSE`) in the error control mechanism.

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

`CV_SUCCESS`	The optional value has been successfully set.
`CVODE_MEM_NULL`	`cvode_mem` is `NULL`.
`CV_NO_SENS`	Sensitivities were not activated.
`CV_NO_QUADSENS`	Quadratures depending on the sensitivities were not activated.

Notes By default, `errconQS` is set to `SUNFALSE`.

It is illegal to call `CVodeSetQuadSensErrCon` before a call to `CVodeQuadSensInit`.

If the quadrature variables are part of the step size control mechanism, one of the following functions must be called to specify the integration tolerances for quadrature variables.

`CVodeQuadSensSStolerances`

Call `flag = CVodeQuadSensSVtolerances(cvode_mem, reltolQS, abstolQS);`

Description The function `CVodeQuadSensSStolerances` specifies scalar relative and absolute tolerances.

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

`reltolQS` (`realtype`) is the scalar relative error tolerance.

`abstolQS` (`realtype*`) is a pointer to an array containing the `Ns` scalar absolute error tolerances.

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

`CV_SUCCESS`	The optional value has been successfully set.
`CVODE_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CV_NO_SENS`	Sensitivities were not activated.
`CV_NO_QUADSENS`	Quadratures depending on the sensitivities were not activated.
`CV_ILL_INPUT`	One of the input tolerances was negative.

`CVodeQuadSensSVtolerances`

Call `flag = CVodeQuadSensSVtolerances(cvode_mem, reltolQS, abstolQS);`

Description The function `CVodeQuadSensSVtolerances` specifies scalar relative and vector absolute tolerances.

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

`reltolQS` (`realtype`) is the scalar relative error tolerance.

`abstolQS` (`N_Vector*`) is an array of `Ns` variables of type `N_Vector`. The `N_Vector abstolS[is]` specifies the vector tolerances for `is`-th quadrature sensitivity.

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

`CV_SUCCESS`	The optional value has been successfully set.
`CV_NO_QUAD`	Quadrature integration was not initialized.

CVODE_MEM_NULL	The cvode_mem pointer is NULL.
CV_NO_SENS	Sensitivities were not activated.
CV_NO_QUADSENS	Quadratures depending on the sensitivities were not activated.
CV_ILL_INPUT	One of the input tolerances was negative.

`CVodeQuadSensEEtolerances`

Call `flag = CVodeQuadSensEEtolerances(cvode_mem);`

Description A call to the function `CVodeQuadSensEEtolerances` specifies that the tolerances for the sensitivity-dependent quadratures should be estimated from those provided for the pure quadrature variables.

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

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

CV_SUCCESS	The optional value has been successfully set.
CVODE_MEM_NULL	The cvode_mem pointer is NULL.
CV_NO_SENS	Sensitivities were not activated.
CV_NO_QUADSENS	Quadratures depending on the sensitivities were not activated.

Notes When `CVodeQuadSensEEtolerances` is used, before calling `CVode`, integration of pure quadratures must be initialized (see 4.7.1) and tolerances for pure quadratures must be also specified (see 4.7.4).

5.4.5 Optional outputs for sensitivity-dependent quadrature integration

CVODES provides the following functions that can be used to obtain solver performance information related to quadrature integration.

`CVodeGetQuadSensNumRhsEvals`

Call `flag = CVodeGetQuadSensNumRhsEvals(cvode_mem, &nrhsQSevals);`

Description The function `CVodeGetQuadSensNumRhsEvals` returns the number of calls made to the user's quadrature right-hand side function.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`nrhsQSevals` (`long int`) number of calls made to the user's `rhsQS` function.

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

CV_SUCCESS	The optional output value has been successfully set.
CVODE_MEM_NULL	The cvode_mem pointer is NULL.
CV_NO_QUADSENS	Sensitivity-dependent quadrature integration has not been initialized.

`CVodeGetQuadSensNumErrTestFails`

Call `flag = CVodeGetQuadSensNumErrTestFails(cvode_mem, &nQSetfails);`

Description The function `CVodeGetQuadSensNumErrTestFails` returns the number of local error test failures due to quadrature variables.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`nQSetfails` (`long int`) number of error test failures due to quadrature variables.

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

CV_SUCCESS	The optional output value has been successfully set.
CVODE_MEM_NULL	The cvode_mem pointer is NULL.
CV_NO_QUADSENS	Sensitivity-dependent quadrature integration has not been initialized.

`CVodeGetQuadSensErrWeights`

Call `flag = CVodeGetQuadSensErrWeights(cvode_mem, eQSweight);`

Description The function `CVodeGetQuadSensErrWeights` returns the quadrature error weights at the current time.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`eQSweight` (`N_Vector *`) array of quadrature error weight vectors 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.
`CVODE_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CV_NO_QUADSENS`	Sensitivity-dependent quadrature integration has not been initialized.

Notes The user must allocate memory for `eQSweight`.

If quadratures were not included in the error control mechanism (through a call to `CVodeSetQuadSensErrCon` with `errconQS = SUNTRUE`), then this function does not set the `eQSweight` array.

`CVodeGetQuadSensStats`

Call `flag = CVodeGetQuadSensStats(cvode_mem, &nrhsQSevals, &nQSetfails);`

Description The function `CVodeGetQuadSensStats` returns the CVODES integrator statistics as a group.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`nrhsQSevals` (`long int`) number of calls to the user's `rhsQS` function.
`nQSetfails` (`long int`) number of error test failures due to quadrature variables.

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

`CV_SUCCESS`	the optional output values have been successfully set.
`CVODE_MEM_NULL`	the `cvode_mem` pointer is `NULL`.
`CV_NO_QUADSENS`	Sensitivity-dependent quadrature integration has not been initialized.

5.4.6 User-supplied function for sensitivity-dependent quadrature integration

For the integration of sensitivity-dependent quadrature equations, the user must provide a function that defines the right-hand side of those quadrature equations. For the sensitivities of quadratures (2.9) with integrand q, the appropriate right-hand side functions are given by: $\bar{q}_i = q_y s_i + q_{p_i}$. This user function must be of type `CVQuadSensRhsFn` defined as follows:

`CVQuadSensRhsFn`

Definition
```
typedef int (*CVQuadSensRhsFn)(int Ns, realtype t, N_Vector y,
                               N_Vector yS, N_Vector yQdot,
                               N_Vector *rhsvalQS, void *user_data,
                               N_Vector tmp1, N_Vector tmp2)
```

Purpose This function computes the sensitivity quadrature equation right-hand side for a given value of the independent variable t and state vector y.

Arguments

`Ns`	is the number of sensitivity vectors.
`t`	is the current value of the independent variable.
`y`	is the current value of the dependent variable vector, $y(t)$.
`yS`	is an array of `Ns` variables of type `N_Vector` containing the dependent sensitivity vectors s_i.

yQdot	is the current value of the quadrature right-hand side, q.
rhsvalQS	array of Ns vectors to contain the right-hand sides.
user_data	is the user_data pointer passed to CVodeSetUserData.
tmp1	
tmp2	are N_Vectors which can be used as temporary storage.

Return value A CVQuadSensRhsFn should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and CV_QRHS_FAIL is returned).

Notes Allocation of memory for rhsvalQS is automatically handled within CVODES.

Here y is of type N_Vector and yS is a pointer to an array containing Ns vectors of type N_Vector. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with CVODES do not perform any consistency checks with respect to their N_Vector arguments (see §7.1 and §7.2).

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

5.5 Note on using partial error control

For some problems, when sensitivities are excluded from the error control test, the behavior of CVODES may appear at first glance to be erroneous. One would expect that, in such cases, the sensitivity variables would not influence in any way the step size selection. A comparison of the solver diagnostics reported for cvsdenx and the second run of the cvsfwddenx example in [35] indicates that this may not always be the case.

The short explanation of this behavior is that the step size selection implemented by the error control mechanism in CVODES is based on the magnitude of the correction calculated by the nonlinear solver. As mentioned in §5.2.1, even with partial error control selected (in the call to CVodeSetSensErrCon), the sensitivity variables are included in the convergence tests of the nonlinear solver.

When using the simultaneous corrector method (§2.6), the nonlinear system that is solved at each step involves both the state and sensitivity equations. In this case, it is easy to see how the sensitivity variables may affect the convergence rate of the nonlinear solver and therefore the step size selection. The case of the staggered corrector approach is more subtle. After all, in this case (ism = CV_STAGGERED or CV_STAGGERED1 in the call to CVodeSensInit/CVodeSensInit1), the sensitivity variables at a given step are computed only once the solver for the nonlinear state equations has converged. However, if the nonlinear system corresponding to the sensitivity equations has convergence problems, CVODES will attempt to improve the initial guess by reducing the step size in order to provide a better prediction of the sensitivity variables. Moreover, even if there are no convergence failures in the solution of the sensitivity system, CVODES may trigger a call to the linear solver's setup routine which typically involves reevaluation of Jacobian information (Jacobian approximation in the case of CVDENSE and CVBAND, or preconditioner data in the case of the Krylov solvers). The new Jacobian information will be used by subsequent calls to the nonlinear solver for the state equations and, in this way, potentially affect the step size selection.

When using the simultaneous corrector method it is not possible to decide whether nonlinear solver convergence failures or calls to the linear solver setup routine have been triggered by convergence problems due to the state or the sensitivity equations. When using one of the staggered corrector methods however, these situations can be identified by carefully monitoring the diagnostic information

provided through optional outputs. If there are no convergence failures in the sensitivity nonlinear solver, and none of the calls to the linear solver setup routine were made by the sensitivity nonlinear solver, then the step size selection is not affected by the sensitivity variables.

Finally, the user must be warned that the effect of appending sensitivity equations to a given system of ODEs on the step size selection (through the mechanisms described above) is problem-dependent and can therefore lead to either an increase or decrease of the total number of steps that CVODES takes to complete the simulation. At first glance, one would expect that the impact of the sensitivity variables, if any, would be in the direction of increasing the step size and therefore reducing the total number of steps. The argument for this is that the presence of the sensitivity variables in the convergence test of the nonlinear solver can only lead to additional iterations (and therefore a smaller final iteration error), or to additional calls to the linear solver setup routine (and therefore more up-to-date Jacobian information), both of which will lead to larger steps being taken by CVODES. However, this is true only locally. Overall, a larger integration step taken at a given time may lead to step size reductions at later times, due to either nonlinear solver convergence failures or error test failures.

Chapter 6

Using CVODES for Adjoint Sensitivity Analysis

This chapter describes the use of CVODES to compute sensitivities of derived functions using adjoint sensitivity analysis. As mentioned before, the adjoint sensitivity module of CVODES provides the infrastructure for integrating backward in time any system of ODEs that depends on the solution of the original IVP, by providing various interfaces to the main CVODES integrator, as well as several supporting user-callable functions. For this reason, in the following sections we refer to the *backward problem* and not to the *adjoint problem* when discussing details relevant to the ODEs that are integrated backward in time. The backward problem can be the adjoint problem (2.19) or (2.22), and can be augmented with some quadrature differential equations.

CVODES 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.

We begin with a brief overview, in the form of a skeleton user program. Following that are detailed descriptions of the interface to the various user-callable functions and of the user-supplied functions that were not already described in Chapter 4.

6.1 A skeleton of the user's main program

The following is a skeleton of the user's main program as an application of CVODES. The user program is to have these steps in the order indicated, unless otherwise noted. For the sake of brevity, we defer many of the details to the later sections. As in §4.4, most steps are independent of the NVECTOR implementation used; where this is not the case, refer to Chapter 7 for specifics. Steps that are unchanged from the skeleton programs presented in §4.4, §5.1, and §5.4, are grayed out.

1. **Include necessary header files**

   The `cvodes.h` header file also defines additional types, constants, and function prototypes for the adjoint sensitivity module user-callable functions. In addition, the main program should include an NVECTOR implementation header file (for the particular implementation used), and, if Newton iteration was selected, the main header file of the desired linear solver module.

2. **Initialize parallel or multi-threaded environment, if appropriate**

Forward problem

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

4. **Set initial conditions for the forward problem**

5. **Create CVODES object for the forward problem**

6. **Initial CVODES for the forward problem**
7. **Specify integration tolerances for forward problem**
8. **Set optional inputs for the forward problem**
9. **Create matrix object for the forward problem**
10. **Create linear solver object for the forward problem**
11. **Set linear solver optional inputs for the forward problem**
12. **Attach linear solver module for the forward problem**
13. **Initialize quadrature problem or problems for forward problems, using `CVodeQuadInit` and/or `CVodeQuadSensInit`.**
14. **Initialize forward sensitivity problem**
15. **Specify rootfinding**
16. **Allocate space for the adjoint computation**

    Call `CVodeAdjInit()` to allocate memory for the combined forward-backward problem (see §6.2.1 for details). This call requires `Nd`, the number of steps between two consecutive checkpoints. `CVodeAdjInit` also specifies the type of interpolation used (see §2.7.1).

17. **Integrate forward problem**

    Call `CVodeF`, a wrapper for the CVODES main integration function `CVode`, either in `CV_NORMAL` mode to the time `tout` or in `CV_ONE_STEP` mode inside a loop (if intermediate solutions of the forward problem are desired (see §6.2.2)). The final value of `tret` is then the maximum allowable value for the endpoint T of the backward problem.

Backward problem(s)

18. **Set problem dimensions etc. for the backward problem**

    This generally includes the backward problem vector length `NB`, and possibly the local vector length `NBlocal`.

19. **Set initial values for the backward problem**

    Set the endpoint time `tB0` $= T$, and set the corresponding vector `yB0` at which the backward problem starts.

20. **Create the backward problem**

    Call `CVodeCreateB`, a wrapper for `CVodeCreate`, to create the CVODES memory block for the new backward problem. Unlike `CVodeCreate`, the function `CVodeCreateB` does not return a pointer to the newly created memory block (see §6.2.3). Instead, this pointer is attached to the internal adjoint memory block (created by `CVodeAdjInit`) and returns an identifier called `which` that the user must later specify in any actions on the newly created backward problem.

21. **Allocate memory for the backward problem**

    Call `CVodeInitB` (or `CVodeInitBS`, when the backward problem depends on the forward sensitivities). The two functions are actually wrappers for `CVodeInit` and allocate internal memory, specify problem data, and initialize CVODES at `tB0` for the backward problem (see §6.2.3).

22. **Specify integration tolerances for backward problem**

Call CVodeSStolerancesB(...) or CVodeSVtolerancesB(...) to specify a scalar relative tolerance and scalar absolute tolerance or scalar relative tolerance and a vector of absolute tolerances, respectively. The functions are wrappers for CVodeSStolerances and CVodeSVtolerances, but they require an extra argument which, the identifier of the backward problem returned by CVodeCreateB. See §6.2.4 for more information.

23. **Set optional inputs for the backward problem**

Call CVodeSet*B functions to change from their default values any optional inputs that control the behavior of CVODES. Unlike their counterparts for the forward problem, these functions take an extra argument which, the identifier of the backward problem returned by CVodeCreateB (see §6.2.8).

24. **Create matrix object for the backward problem**

If a direct linear solver is to be used within a Newton iteration then a template Jacobian matrix must be created by using the appropriate functions defined by the particular SUNMATRIX implementation.

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

Note also that it is not required to use the same matrix type for both the forward and the backward problems.

25. **Create linear solver object for the backward problem**

Create the linear solver object for the backward problem by using the appropriate functions defined by the particular SUNLINSOL implementation desired.

Note that it is not required to use the same linear solver module for both the forward and the backward problems; for example, the forward problem could be solved with the CVDLS linear solver module and the backward problem with CVSPILS linear solver module.

26. **Set linear solver interface optional inputs for the backward problem**

Call CVDlsSet*B or CVSpilsSet*B functions to change optional inputs specific to that linear solver interface. See §6.2.8 for details.

27. **Initialize quadrature calculation**

If additional quadrature equations must be evaluated, call CVodeQuadInitB or CVodeQuadInitBS (if quadrature depends also on the forward sensitivities) as shown in §6.2.10.1. These functions are wrappers around CVodeQuadInit and can be used to initialize and allocate memory for quadrature integration. Optionally, call CVodeSetQuad*B functions to change from their default values optional inputs that control the integration of quadratures during the backward phase.

28. **Integrate backward problem**

Call CVodeB, a second wrapper around the CVODES main integration function CVode, to integrate the backward problem from tB0 (see §6.2.6). This function can be called either in CV_NORMAL or CV_ONE_STEP mode. Typically, CVodeB will be called in CV_NORMAL mode with an end time equal to the initial time t_0 of the forward problem.

29. **Extract quadrature variables**

If applicable, call CVodeGetQuadB, a wrapper around CVodeGetQuad, to extract the values of the quadrature variables at the time returned by the last call to CVodeB. See §6.2.10.2.

30. **Deallocate memory**

Upon completion of the backward integration, call all necessary deallocation functions. These include appropriate destructors for the vectors y and yB, a call to CVodeFree to free the CVODES

memory block for the forward problem. If one or more additional Adjoint Sensitivity Analyses are to be done for this problem, a call to `CVodeAdjFree` (see §6.2.1) may be made to free and deallocate memory allocated for the backward problems, followed by a call to `CVodeAdjInit`.

31. **Free linear solver and matrix memory for the backward problem**

32. **Finalize MPI, if used**

The above user interface to the adjoint sensitivity module in CVODES was motivated by the desire to keep it as close as possible in look and feel to the one for ODE IVP integration. Note that if steps (18)-(29) are not present, a program with the above structure will have the same functionality as one described in §4.4 for integration of ODEs, albeit with some overhead due to the checkpointing scheme.

If there are multiple backward problems associated with the same forward problem, repeat steps (18)-(29) above for each successive backward problem. In the process, each call to `CVodeCreateB` creates a new value of the identifier `which`.

6.2 User-callable functions for adjoint sensitivity analysis

6.2.1 Adjoint sensitivity allocation and deallocation functions

After the setup phase for the forward problem, but before the call to `CVodeF`, memory for the combined forward-backward problem must be allocated by a call to the function `CVodeAdjInit`. The form of the call to this function is

`CVodeAdjInit`

Call `flag = CVodeAdjInit(cvode_mem, Nd, interpType);`

Description The function `CVodeAdjInit` updates CVODES memory block by allocating the internal memory needed for backward integration. Space is allocated for the `Nd` $= N_d$ interpolation data points, and a linked list of checkpoints is initialized.

Arguments

`cvode_mem`	(`void *`) is the pointer to the CVODES memory block returned by a previous call to `CVodeCreate`.
`Nd`	(`long int`) is the number of integration steps between two consecutive checkpoints.
`interpType`	(`int`) specifies the type of interpolation used and can be `CV_POLYNOMIAL` or `CV_HERMITE`, indicating variable-degree polynomial and cubic Hermite interpolation, respectively (see §2.7.1).

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

`CV_SUCCESS`	`CVodeAdjInit` was successful.
`CV_MEM_FAIL`	A memory allocation request has failed.
`CV_MEM_NULL`	`cvode_mem` was NULL.
`CV_ILL_INPUT`	One of the parameters was invalid: `Nd` was not positive or `interpType` is not one of the `CV_POLYNOMIAL` or `CV_HERMITE`.

Notes The user must set `Nd` so that all data needed for interpolation of the forward problem solution between two checkpoints fits in memory. `CVodeAdjInit` attempts to allocate space for (2`Nd`+3) variables of type `N_Vector`.

If an error occurred, `CVodeAdjInit` also sends a message to the error handler function.

`CVodeAdjReInit`

Call `flag = CVodeAdjReInit(cvode_mem);`

Description The function `CVodeAdjReInit` reinitializes the CVODES memory block for ASA, assuming that the number of steps between check points and the type of interpolation remain unchanged.

Arguments `cvode_mem` (`void *`) is the pointer to the CVODES memory block returned by a previous call to `CVodeCreate`.

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

`CV_SUCCESS` `CVodeAdjReInit` was successful.

`CV_MEM_NULL` `cvode_mem` was NULL.

`CV_NO_ADJ` The function `CVodeAdjInit` was not previously called.

Notes The list of check points (and associated memory) is deleted.

The list of backward problems is kept. However, new backward problems can be added to this list by calling `CVodeCreateB`. If a new list of backward problems is also needed, then free the adjoint memory (by calling `CVodeAdjFree`) and reinitialize ASA with `CVodeAdjInit`.

The CVODES memory for the forward and backward problems can be reinitialized separately by calling `CVodeReInit` and `CVodeReInitB`, respectively.

`CVodeAdjFree`

Call `CVodeAdjFree(cvode_mem);`

Description The function `CVodeAdjFree` frees the memory related to backward integration allocated by a previous call to `CVodeAdjInit`.

Arguments The only argument is the CVODES memory block pointer returned by a previous call to `CVodeCreate`.

Return value The function `CVodeAdjFree` has no return value.

Notes This function frees all memory allocated by `CVodeAdjInit`. This includes workspace memory, the linked list of checkpoints, memory for the interpolation data, as well as the CVODES memory for the backward integration phase. Unless one or more further calls to `CVodeAdjInit` are to be made, `CVodeAdjFree` should not be called by the user, as it is invoked automatically by `CVodeFree`.

6.2.2 Forward integration function

The function `CVodeF` is very similar to the CVODES function `CVode` (see §4.5.5) in that it integrates the solution of the forward problem and returns the solution in `y`. At the same time, however, `CVodeF` stores checkpoint data every `Nd` integration steps. `CVodeF` can be called repeatedly by the user. Note that `CVodeF` is used only for the forward integration pass within an Adjoint Sensitivity Analysis. It is not for use in Forward Sensitivity Analysis; for that, see Chapter 5. The call to this function has the form

`CVodeF`

Call `flag = CVodeF(cvode_mem, tout, yret, &tret, itask, &ncheck);`

Description The function `CVodeF` integrates the forward problem over an interval in t and saves checkpointing data.

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

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

`yret` (`N_Vector`) the computed solution vector y.

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

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

`ncheck` (`int`) the number of (internal) checkpoints stored so far.

Return value On return, `CVodeF` returns the vector `yret` and a corresponding independent variable value $t =$ `tret`, such that `yret` is the computed value of $y(t)$. Additionally, it returns in `ncheck` the number of internal checkpoints saved; the total number of checkpoint intervals is `ncheck`+1. The return value `flag` (of type `int`) will be one of the following. For more details see §4.5.5.

`CV_SUCCESS`	`CVodeF` succeeded.
`CV_TSTOP_RETURN`	`CVodeF` succeeded by reaching the optional stopping point.
`CV_ROOT_RETURN`	`CVodeF` succeeded and found one or more roots. In this case, `tret` is the location of the root. If `nrtfn` > 1, call `CVodeGetRootInfo` to see which g_i were found to have a root.
`CV_NO_MALLOC`	The function `CVodeInit` has not been previously called.
`CV_ILL_INPUT`	One of the inputs to `CVodeF` is illegal.
`CV_TOO_MUCH_WORK`	The solver took `mxstep` internal steps but could not reach `tout`.
`CV_TOO_MUCH_ACC`	The solver could not satisfy the accuracy demanded by the user for some internal step.
`CV_ERR_FAILURE`	Error test failures occurred too many times during one internal time step or occurred with $\|h\| = h_{min}$.
`CV_CONV_FAILURE`	Convergence test failures occurred too many times during one internal time step or occurred with $\|h\| = h_{min}$.
`CV_LSETUP_FAIL`	The linear solver's setup function failed in an unrecoverable manner.
`CV_LSOLVE_FAIL`	The linear solver's solve function failed in an unrecoverable manner.
`CV_NO_ADJ`	The function `CVodeAdjInit` has not been previously called.
`CV_MEM_FAIL`	A memory allocation request has failed (in an attempt to allocate space for a new checkpoint).

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

At this time, `CVodeF` stores checkpoint information in memory only. Future versions will provide for a safeguard option of dumping checkpoint data into a temporary file as needed. The data stored at each checkpoint is basically a snapshot of the CVODES internal memory block and contains enough information to restart the integration from that time and to proceed with the same step size and method order sequence as during the forward integration.

In addition, `CVodeF` also stores interpolation data between consecutive checkpoints so that, at the end of this first forward integration phase, interpolation information is already available from the last checkpoint forward. In particular, if no checkpoints were necessary, there is no need for the second forward integration phase.

It is illegal to change the integration tolerances between consecutive calls to `CVodeF`, as this information is not captured in the checkpoint data.

6.2.3 Backward problem initialization functions

The functions `CVodeCreateB` and `CVodeInitB` (or `CVodeInitBS`) must be called in the order listed. They instantiate a CVODES solver object, provide problem and solution specifications, and allocate internal memory for the backward problem.

`CVodeCreateB`

Call `flag = CVodeCreateB(cvode_mem, lmmB, iterB, &which);`

Description The function `CVodeCreateB` instantiates a CVODES solver object and specifies the solution method for the backward problem.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block returned by `CVodeCreate`.
- `lmmB` (`int`) specifies the linear multistep method and may be one of two possible values: `CV_ADAMS` or `CV_BDF`.
- `iterB` (`int`) specifies the type of nonlinear solver iteration and may be either `CV_NEWTON` or `CV_FUNCTIONAL`.
- `which` (`int`) contains the identifier assigned by CVODES for the newly created backward problem. Any call to `CVode*B` functions requires such an identifier.

Return value The return value `flag` (of type `int`) is one of:
- `CV_SUCCESS` The call to `CVodeCreateB` was successful.
- `CV_MEM_NULL` `cvode_mem` was `NULL`.
- `CV_NO_ADJ` The function `CVodeAdjInit` has not been previously called.
- `CV_MEM_FAIL` A memory allocation request has failed.

There are two initialization functions for the backward problem – one for the case when the backward problem does not depend on the forward sensitivities, and one for the case when it does. These two functions are described next.

The function `CVodeInitB` initializes the backward problem when it does not depend on the forward sensitivities. It is essentially a wrapper for `CVodeInit` with some particularization for backward integration, as described below.

`CVodeInitB`

Call `flag = CVodeInitB(cvode_mem, which, rhsB, tB0, yB0);`

Description The function `CVodeInitB` provides problem specification, allocates internal memory, and initializes the backward problem.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block returned by `CVodeCreate`.
- `which` (`int`) represents the identifier of the backward problem.
- `rhsB` (`CVRhsFnB`) is the C function which computes fB, the right-hand side of the backward ODE problem. This function has the form `rhsB(t, y, yB, yBdot, user_dataB)` (for full details see §6.3.1).
- `tB0` (`realtype`) specifies the endpoint T where final conditions are provided for the backward problem, normally equal to the endpoint of the forward integration.
- `yB0` (`N_Vector`) is the initial value (at $t =$ `tB0`) of the backward solution.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeInitB` was successful.
- `CV_NO_MALLOC` The function `CVodeInit` has not been previously called.
- `CV_MEM_NULL` `cvode_mem` was `NULL`.
- `CV_NO_ADJ` The function `CVodeAdjInit` has not been previously called.
- `CV_BAD_TB0` The final time `tB0` was outside the interval over which the forward problem was solved.
- `CV_ILL_INPUT` The parameter `which` represented an invalid identifier, or either `yB0` or `rhsB` was `NULL`.

Notes The memory allocated by `CVodeInitB` is deallocated by the function `CVodeAdjFree`.

For the case when backward problem also depends on the forward sensitivities, user must call `CVodeInitBS` instead of `CVodeInitB`. Only the third argument of each function differs between these two functions.

`CVodeInitBS`

Call `flag = CVodeInitBS(cvode_mem, which, rhsBS, tB0, yB0);`

Description The function `CVodeInitBS` provides problem specification, allocates internal memory, and initializes the backward problem.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block returned by `CVodeCreate`.
- `which` (`int`) represents the identifier of the backward problem.
- `rhsBS` (`CVRhsFnBS`) is the C function which computes fB, the right-hand side of the backward ODE problem. This function has the form `rhsBS(t, y, yS, yB, yBdot, user_dataB)` (for full details see §6.3.2).
- `tB0` (`realtype`) specifies the endpoint T where final conditions are provided for the backward problem.
- `yB0` (`N_Vector`) is the initial value (at $t =$ `tB0`) of the backward solution.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeInitB` was successful.
- `CV_NO_MALLOC` The function `CVodeInit` has not been previously called.
- `CV_MEM_NULL` `cvode_mem` was `NULL`.
- `CV_NO_ADJ` The function `CVodeAdjInit` has not been previously called.
- `CV_BAD_TB0` The final time `tB0` was outside the interval over which the forward problem was solved.
- `CV_ILL_INPUT` The parameter `which` represented an invalid identifier, either `yB0` or `rhsBS` was `NULL`, or sensitivities were not active during the forward integration.

Notes The memory allocated by `CVodeInitBS` is deallocated by the function `CVodeAdjFree`.

The function `CVodeReInitB` reinitializes CVODES for the solution of a series of backward problems, each identified by a value of the parameter `which`. `CVodeReInitB` is essentially a wrapper for `CVodeReInit`, and so all details given for `CVodeReInit` in §4.5.9 apply here. Also note that `CVodeReInitB` can be called to reinitialize the backward problem even it has been initialized with the sensitivity-dependent version `CVodeInitBS`. Before calling `CVodeReInitB` for a new backward problem, call any desired solution extraction functions `CVodeGet**` associated with the previous backward problem. The call to the `CVodeReInitB` function has the form

`CVodeReInitB`

Call `flag = CVodeReInitB(cvode_mem, which, tB0, yB0)`

Description The function `CVodeReInitB` reinitializes a CVODES backward problem.

Arguments
- `cvode_mem` (`void *`) pointer to CVODES memory block returned by `CVodeCreate`.
- `which` (`int`) represents the identifier of the backward problem.
- `tB0` (`realtype`) specifies the endpoint T where final conditions are provided for the backward problem.
- `yB0` (`N_Vector`) is the initial value (at $t =$ `tB0`) of the backward solution.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeReInitB` was successful.
- `CV_NO_MALLOC` The function `CVodeInit` has not been previously called.
- `CV_MEM_NULL` The `cvode_mem` memory block pointer was `NULL`.
- `CV_NO_ADJ` The function `CVodeAdjInit` has not been previously called.
- `CV_BAD_TB0` The final time `tB0` is outside the interval over which the forward problem was solved.
- `CV_ILL_INPUT` The parameter `which` represented an invalid identifier, or `yB0` was `NULL`.

6.2.4 Tolerance specification functions for backward problem

One of the following two functions must be called to specify the integration tolerances for the backward problem. Note that this call must be made after the call to `CVodeInitB` or `CVodeInitBS`.

`CVodeSStolerancesB`

Call `flag = CVodeSStolerancesB(cvode_mem, which, reltolB, abstolB);`

Description The function `CVodeSStolerancesB` specifies scalar relative and absolute tolerances.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block returned by `CVodeCreate`.
- `which` (`int`) represents the identifier of the backward problem.
- `reltolB` (`realtype`) is the scalar relative error tolerance.
- `abstolB` (`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 `CVodeSStolerancesB` was successful.
- `CV_MEM_NULL` The CVODES memory block was not initialized through a previous call to `CVodeCreate`.
- `CV_NO_MALLOC` The allocation function `CVodeInit` has not been called.
- `CV_NO_ADJ` The function `CVodeAdjInit` has not been previously called.
- `CV_ILL_INPUT` One of the input tolerances was negative.

`CVodeSVtolerancesB`

Call `flag = CVodeSVtolerancesB(cvode_mem, which, reltolB, abstolB);`

Description The function `CVodeSVtolerancesB` specifies scalar relative tolerance and vector absolute tolerances.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block returned by `CVodeCreate`.
- `which` (`int`) represents the identifier of the backward problem.
- `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 `CVodeSVtolerancesB` was successful.
- `CV_MEM_NULL` The CVODES memory block was not initialized through a previous call to `CVodeCreate`.
- `CV_NO_MALLOC` The allocation function `CVodeInit` has not been called.
- `CV_NO_ADJ` The function `CVodeAdjInit` has not been previously 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.

6.2.5 Linear solver initialization functions for backward problem

All CVODES linear solver modules available for forward problems are available for the backward problem. They should be created as for the forward problem then attached to the memory structure for the backward problem using one of the following functions.

`CVDlsSetLinearSolverB`

Call `flag = CVDlsSetLinearSolverB(cvode_mem, which, LS, A);`

Description The function `CVDlsSetLinearSolverB` attaches a direct SUNLINSOL object `LS` and corresponding template Jacobian SUNMATRIX object `A` to CVODES, initializing the CVDLS direct linear solver interface for solution of the backward problem.

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

Arguments
- `cvode_mem` (`void *`) pointer to the IDAS memory block.
- `which` (`int`) represents the identifier of the backward problem returned by `CVodeCreateB`.
- `LS` (`SUNLinearSolver`) SUNLINSOL object to use for solving Newton linear systems for the backward problem.
- `A` (`SUNMatrix`) SUNMATRIX object for used as a template for the Jacobian for the backward problem (must have a type compatible with the linear solver object).

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

- `CVDLS_SUCCESS` The CVDLS initialization was successful.
- `CVDLS_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVDLS_ILL_INPUT` The CVDLS solver is not compatible with the current NVECTOR module.
- `CVDLS_MEM_FAIL` A memory allocation request failed.
- `CVDLS_NO_ADJ` The function `CVAdjInit` has not been previously called.
- `CVDLS_ILL_INPUT` The parameter `which` represented an invalid identifier.

Notes The CVDLS linear solver is not compatible with all implementations of the SUNLINSOL and NVECTOR modules. Specifically, CVDLS requires use of a *direct* SUNLINSOL object and a serial or theaded NVECTOR module. Additional compatibility limitations for each SUNLINSOL object (i.e. SUNMATRIX and NVECTOR object compatibility) are described in Chapter 9.

`CVSpilsSetLinearSolverB`

Call `flag = CVSpilsSetLinearSolverB(ida_mem, which, LS);`

Description The function `CVSpilsSetLinearSolver` attaches an iterative SUNLINSOL object `LS` to CVODES, initializing the CVSPILS scaled, preconditioned, iterative linear solver interface to use for the backward problem.

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `which` (`int`) represents the identifier of the backward problem returned by `CVodeCreateB`.
- `LS` (`SUNLinearSolver`) SUNLINSOL object to use for solving Newton linear systems for the backward problem.

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

- `CVSPILS_SUCCESS` The CVSPILS initialization was successful.
- `CVSPILS_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVSPILS_ILL_INPUT` The CVSPILS solver is not compatible with the current NVECTOR module.
- `CVSPILS_MEM_FAIL` A memory allocation request failed.
- `CVSPILS_NO_ADJ` The function `CVAdjInit` has not been previously called.
- `CVSPILS_ILL_INPUT` The parameter `which` represented an invalid identifier.

Notes The CVSPILS linear solver interface is not compatible with all implementations of the SUNLINSOL and NVECTOR modules. Specifically, CVSPILS requires use of an *iterative* SUNLINSOL object. Additional compatibility limitations for each SUNLINSOL object (i.e. required NVECTOR routines) are described in Chapter 9.

6.2.6 Backward integration function

The function `CVodeB` performs the integration of the backward problem. It is essentially a wrapper for the CVODES main integration function `CVode` and, in the case in which checkpoints were needed, it evolves the solution of the backward problem through a sequence of forward-backward integration pairs between consecutive checkpoints. The first run of each pair integrates the original IVP forward in time and stores interpolation data; the second run integrates the backward problem backward in time and performs the required interpolation to provide the solution of the IVP to the backward problem.

The function `CVodeB` does not return the solution `yB` itself. To obtain that, call the function `CVodeGetB`, which is also described below.

The `CVodeB` function does not support rootfinding, unlike `CVodeF`, which supports the finding of roots of functions of (t, y). If rootfinding was performed by `CVodeF`, then for the sake of efficiency, it should be disabled for `CVodeB` by first calling `CVodeRootInit` with `nrtfn` $= 0$.

The call to `CVodeB` has the form

`CVodeB`

Call `flag = CVodeB(cvode_mem, tBout, itaskB);`

Description The function `CVodeB` integrates the backward ODE problem.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory returned by `CVodeCreate`.
- `tBout` (`realtype`) the next time at which a computed solution is desired.
- `itaskB` (`int`) a flag indicating the job of the solver for the next step. The `CV_NORMAL` task is to have the solver take internal steps until it has reached or just passed the user-specified value `tBout`. The solver then interpolates in order to return an approximate value of yB(`tBout`). The `CV_ONE_STEP` option tells the solver to take just one internal step in the direction of `tBout` and return.

Return value The return value `flag` (of type `int`) will be one of the following. For more details see §4.5.5.

`CV_SUCCESS`	`CVodeB` succeeded.
`CV_MEM_NULL`	`cvode_mem` was `NULL`.
`CV_NO_ADJ`	The function `CVodeAdjInit` has not been previously called.
`CV_NO_BCK`	No backward problem has been added to the list of backward problems by a call to `CVodeCreateB`
`CV_NO_FWD`	The function `CVodeF` has not been previously called.
`CV_ILL_INPUT`	One of the inputs to `CVodeB` is illegal.
`CV_BAD_ITASK`	The `itaskB` argument has an illegal value.
`CV_TOO_MUCH_WORK`	The solver took `mxstep` internal steps but could not reach `tBout`.
`CV_TOO_MUCH_ACC`	The solver could not satisfy the accuracy demanded by the user for some internal step.
`CV_ERR_FAILURE`	Error test failures occurred too many times during one internal time step.
`CV_CONV_FAILURE`	Convergence test failures occurred too many times during one internal time step.
`CV_LSETUP_FAIL`	The linear solver's setup function failed in an unrecoverable manner.
`CV_SOLVE_FAIL`	The linear solver's solve function failed in an unrecoverable manner.

CV_BCKMEM_NULL	The solver memory for the backward problem was not created with a call to CVodeCreateB.
CV_BAD_TBOUT	The desired output time tBout is outside the interval over which the forward problem was solved.
CV_REIFWD_FAIL	Reinitialization of the forward problem failed at the first checkpoint (corresponding to the initial time of the forward problem).
CV_FWD_FAIL	An error occurred during the integration of the forward problem.

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

In the case of multiple checkpoints and multiple backward problems, a given call to CVodeB in CV_ONE_STEP mode may not advance every problem one step, depending on the relative locations of the current times reached. But repeated calls will eventually advance all problems to tBout.

To obtain the solution yB to the backward problem, call the function CVodeGetB as follows:

CVodeGetB

Call `flag = CVodeGetB(cvode_mem, which, &tret, yB);`

Description The function CVodeGetB provides the solution yB of the backward ODE problem.

Arguments

cvode_mem	(void *) pointer to the CVODES memory returned by CVodeCreate.
which	(int) the identifier of the backward problem.
tret	(realtype) the time reached by the solver (output).
yB	(N_Vector) the backward solution at time tret.

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

CV_SUCCESS	CVodeGetB was successful.
CV_MEM_NULL	cvode_mem is NULL.
CV_NO_ADJ	The function CVodeAdjInit has not been previously called.
CV_ILL_INPUT	The parameter which is an invalid identifier.

Notes The user must allocate space for yB.

6.2.7 Adjoint sensitivity optional input

At any time during the integration of the forward problem, the user can disable the checkpointing of the forward sensitivities by calling the following function:

CVodeAdjSetNoSensi

Call `flag = CVodeAdjSetNoSensi(cvode_mem);`

Description The function CVodeAdjSetNoSensi instructs CVodeF not to save checkpointing data for forward sensitivities anymore.

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

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

CV_SUCCESS	The call to CVodeCreateB was successful.
CV_MEM_NULL	cvode_mem was NULL.
CV_NO_ADJ	The function CVodeAdjInit has not been previously called.

6.2.8 Optional input functions for the backward problem

6.2.8.1 Main solver optional input functions

The adjoint module in CVODES provides wrappers for most of the optional input functions defined in §4.5.6.1. The only difference is that the user must specify the identifier `which` of the backward problem within the list managed by CVODES.

The optional input functions defined for the backward problem are:

```
flag = CVodeSetUserDataB(cvode_mem, which, user_dataB);
flag = CVodeSetIterTypeB(cvode_mem, which, iterB);
flag = CVodeSetMaxOrdB(cvode_mem, which, maxordB);
flag = CVodeSetMaxNumStepsB(cvode_mem, which, mxstepsB);
flag = CVodeSetInitStepB(cvode_mem, which, hinB)
flag = CVodeSetMinStepB(cvode_mem, which, hminB);
flag = CVodeSetMaxStepB(cvode_mem, which, hmaxB);
flag = CVodeSetStabLimDetB(cvode_mem, which, stldetB);
flag = CVodeSetConstraintsB(cvode_mem, which, constraintsB);
```

Their return value `flag` (of type `int`) can have any of the return values of their counterparts, but it can also be `CV_NO_ADJ` if `CVodeAdjInit` has not been called, or `CV_ILL_INPUT` if `which` was an invalid identifier.

6.2.8.2 Direct linear solver interface optional input functions

If using a direct linear solver interface for the Jacobian of the backward problem, the linear solver will need to be attached to the memory structure through a call to `CVDlsSetLinearSolverB`. The Jacobian evaluation function can be attached through a call to either `CVDlsSetJacFnB` or IDACVDlsSetJacFnBS, with the second used when the backward problem depends on the forwrad sensitivities.

`CVDlsSetJacFnB`

Call `flag = CVDlsSetJacFnB(ida_mem, which, jacB);`

Description The function `CVDlsSetJacFnB` specifies the Jacobian approximation function to be used for the backward problem.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory returned by `CVodeCreate`.
- `which` (`int`) represents the identifier of the backward problem.
- `jacB` (`CVDlsJacFnB`) user-defined Jacobian approximation function.

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

`CVDLS_SUCCESS`	`CVDlsSetJacFnB` succeeded.
`CVDLS_MEM_NULL`	`cvode_mem` was `NULL`.
`CVDLS_NO_ADJ`	The function `CVodeAdjInit` has not been previously called.
`CVDLS_LMEM_NULL`	The linear solver has not been initialized with a call to `CVDlsSetLinearSolverB`.
`CVDLS_ILL_INPUT`	The parameter `which` represented an invalid identifier.

Notes The function type `CVDlsJacFnB` is described in §6.3.5.

`CVDlsSetJacFnBS`

Call `flag = CVDlsSetJacFnBS(cvode_mem, which, jacBS);`

Description The function `CVDlsSetJacFnBS` specifies the Jacobian approximation function to be used for the backward problem, in the case where the backward problem depends on the forward sensitivities.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory returned by `CVodeCreate`.

`which` (`int`) represents the identifier of the backward problem.
`jacBS` (`CVDlsJacFnBS`) user-defined Jacobian approximation function.

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

`CVDLS_SUCCESS`	`CVDlsSetJacFnBS` succeeded.
`CVDLS_MEM_NULL`	`cvode_mem` was `NULL`.
`CVDLS_NO_ADJ`	The function `CVodeAdjInit` has not been previously called.
`CVDLS_LMEM_NULL`	The linear solver has not been initialized with a call to `CVDlsSetLinearSolverB`.
`CVDLS_ILL_INPUT`	The parameter `which` represented an invalid identifier.

Notes The function type `CVDlsJacFnBS` is described in §6.3.5.

6.2.8.3 SPILS linear solvers

Optional inputs for the CVSPILS linear solver module can be set for the backward problem through the following functions:

`CVSpilsSetPreconditionerB`

Call `flag = CVSpilsSetPreconditionerB(cvode_mem, which, psetupB, psolveB);`

Description The function `CVSpilsSetPrecSolveFnB` specifies the preconditioner setup and solve functions for the backward integration.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`which` (`int`) the identifier of the backward problem.
`psetupB` (`CVSpilsPrecSetupFnB`) user-defined preconditioner setup function.
`psolveB` (`CVSpilsPrecSolveFnB`) user-defined preconditioner solve function.

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

`CVSPILS_SUCCESS`	The optional value has been successfully set.
`CVSPILS_MEM_NULL`	`cvode_mem` was `NULL`.
`CVSPILS_LMEM_NULL`	The CVSPILS linear solver has not been initialized.
`CVSPILS_NO_ADJ`	The function `CVodeAdjInit` has not been previously called.
`CVSPILS_ILL_INPUT`	The parameter `which` represented an invalid identifier.

Notes The function types `CVSpilsPrecSolveFnB` and `CVSpilsPrecSetupFnB` are described in §6.3.8 and §6.3.9, resp. The `psetupB` argument may be `NULL` if no setup operation is involved in the preconditioner.

`CVSpilsSetPreconditionerBS`

Call `flag = CVSpilsSetPreconditionerBS(cvode_mem, which, psetupBS, psolveBS);`

Description The function `CVSpilsSetPrecSolveFnBS` specifies the preconditioner setup and solve functions for the backward integration, in the case where the backward problem depends on the forward sensitivities.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`which` (`int`) the identifier of the backward problem.
`psetupBS` (`CVSpilsPrecSetupFnBS`) user-defined preconditioner setup function.
`psolveBS` (`CVSpilsPrecSolveFnBS`) user-defined preconditioner solve function.

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

`CVSPILS_SUCCESS`	The optional value has been successfully set.
`CVSPILS_MEM_NULL`	`cvode_mem` was `NULL`.
`CVSPILS_LMEM_NULL`	The CVSPILS linear solver has not been initialized.

CVSPILS_NO_ADJ	The function CVodeAdjInit has not been previously called.
CVSPILS_ILL_INPUT	The parameter which represented an invalid identifier.

Notes The function types CVSpilsPrecSolveFnBS and CVSpilsPrecSetupFnBS are described in §6.3.8 and §6.3.9, resp. The psetupBS argument may be NULL if no setup operation is involved in the preconditioner.

CVSpilsSetJacTimesB

Call `flag = CVSpilsSetJacTimesB(cvode_mem, which, jsetupB, jtvB);`

Description The function CVSpilsSetJacTimesB specifies the Jacobian-vector setup and product functions to be used.

Arguments

cvode_mem	(void *) pointer to the CVODES memory block.
which	(int) the identifier of the backward problem.
jtsetupB	(CVSpilsJacTimesSetupFnB) user-defined function to set up the Jacobian-vector product. Pass NULL if no setup is necessary.
jtvB	(CVSpilsJacTimesVecFnB) user-defined Jacobian-vector product function.

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

CVSPILS_SUCCESS	The optional value has been successfully set.
CVSPILS_MEM_NULL	cvode_mem was NULL.
CVSPILS_LMEM_NULL	The CVSPILS linear solver has not been initialized.
CVSPILS_NO_ADJ	The function CVodeAdjInit has not been previously called.
CVSPILS_ILL_INPUT	The parameter which represented an invalid identifier.

Notes The function types CVSpilsJacTimesVecFnB and CVSpilsJacTimesSetupFnB are described in §6.3.6.

CVSpilsSetJacTimesBS

Call `flag = CVSpilsSetJacTimesBS(cvode_mem, which, jtvBS);`

Description The function CVSpilsSetJacTimesBS specifies the Jacobian-vector setup and product functions to be used, in the case where the backward problem depends on the forward sensitivities.

Arguments

cvode_mem	(void *) pointer to the CVODES memory block.
which	(int) the identifier of the backward problem.
jtsetupBS	(CVSpilsJacTimesSetupFnBS) user-defined function to set up the Jacobian-vector product. Pass NULL if no setup is necessary.
jtvBS	(CVSpilsJacTimesVecFnBS) user-defined Jacobian-vector product function.

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

CVSPILS_SUCCESS	The optional value has been successfully set.
CVSPILS_MEM_NULL	cvode_mem was NULL.
CVSPILS_LMEM_NULL	The CVSPILS linear solver has not been initialized.
CVSPILS_NO_ADJ	The function CVodeAdjInit has not been previously called.
CVSPILS_ILL_INPUT	The parameter which represented an invalid identifier.

Notes The function types CVSpilsJacTimesVecFnBS and CVSpilsJacTimesSetupFnBS are described in §6.3.6.

`CVSpilsSetEpsLinB`

Call `flag = CVSpilsSetEpsLinB(cvode_mem, which, eplifacB);`

Description The function `CVSpilsSetEpsLinB` specifies the factor by which the Krylov linear solver's convergence test constant is reduced from the Newton iteration test constant. This routine can be used in both the cases wherethe backward problem does and does not depend on the forward sensitvities.

Arguments
`cvode_mem` (`void *`) pointer to the CVODES memory block.
`which` (`int`) the identifier of the backward problem.
`eplifacB` (`realtype`) value of the convergence test constant reduction factor (≥ 0.0).

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

`CVSPILS_SUCCESS`	The optional value has been successfully set.
`CVSPILS_MEM_NULL`	`cvode_mem` was `NULL`.
`CVSPILS_LMEM_NULL`	The CVSPILS linear solver has not been initialized.
`CVSPILS_NO_ADJ`	The function `CVodeAdjInit` has not been previously called.
`CVSPILS_ILL_INPUT`	The parameter `which` represented an invalid identifier, or `eplifacB` was negative.

Notes The default value is 0.05. Passing a value `eplifacB`$= 0.0$ also indicates using the default value.

6.2.9 Optional output functions for the backward problem

The user of the adjoint module in CVODES has access to any of the optional output functions described in §4.5.8, both for the main solver and for the linear solver modules. The first argument of these `CVodeGet*` and `CVode*Get*` functions is the pointer to the CVODES memory block for the backward problem. In order to call any of these functions, the user must first call the following function to obtain this pointer.

`CVodeGetAdjCVodeBmem`

Call `cvode_memB = CVodeGetAdjCVodeBmem(cvode_mem, which);`

Description The function `CVodeGetAdjCVodeBmem` returns a pointer to the CVODES memory block for the backward problem.

Arguments
`cvode_mem` (`void *`) pointer to the CVODES memory block created by `CVodeCreate`.
`which` (`int`) the identifier of the backward problem.

Return value The return value, `cvode_memB` (of type `void *`), is a pointer to the CVODES memory for the backward problem.

Notes The user should not modify `cvode_memB` in any way.

Optional output calls should pass `cvode_memB` as the first argument; for example, to get the number of integration steps: `flag = CVodeGetNumSteps(cvodes_memB, &nsteps)`.

To get values of the *forward* solution during a backward integration, use the following function. The input value of `t` would typically be equal to that at which the backward solution has just been obtained with `CVodeGetB`. In any case, it must be within the last checkpoint interval used by `CVodeB`.

`CVodeGetAdjY`

Call `flag = CVodeGetAdjY(cvode_mem, t, y);`

Description The function `CVodeGetAdjY` returns the interpolated value of the forward solution y during a backward integration.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block created by `CVodeCreate`.

	`t`	(`realtype`) value of the independent variable at which y is desired (input).
	`y`	(`N_Vector`) forward solution $y(t)$.
Return value	The return value `flag` (of type `int`) is one of:	
	`CV_SUCCESS`	`CVodeGetAdjY` was successful.
	`CV_MEM_NULL`	`cvode_mem` was `NULL`.
	`CV_GETY_BADT`	The value of `t` was outside the current checkpoint interval.
Notes	The user must allocate space for `y`.	

`CVodeGetAdjCheckPointsInfo`

Call `flag = CVodeGetAdjCheckPointsInfo(cvode_mem, CVadjCheckPointRec *ckpnt);`

Description The function `CVodeGetAdjCheckPointsInfo` loads an array of `ncheck+1` records of type `CVadjCheckPointRec`. The user must allocate space for the array `ckpnt`.

Arguments

`cvode_mem`	(`void *`) pointer to the CVODES memory block created by `CVodeCreate`.
`ckpnt`	(`CVadjCheckPointRec *`) array of `ncheck+1` checkpoint records, each of type `CVadjCheckPointRec`.

Return value The return value is `CV_SUCCESS` if successful, or `CV_MEM_NULL` if `cvode_mem` is `NULL`, or `CV_NO_ADJ` if ASA was not initialized.

Notes The members of each record `ckpnt[i]` are:

- : `ckpnt[i].my_addr` (`void *`) address of current checkpoint in `cvode_mem->cv_adj_mem`
- : `ckpnt[i].next_addr` (`void *`) address of next checkpoint
- : `ckpnt[i].t0` (`realtype`) start of checkpoint interval
- : `ckpnt[i].t1` (`realtype`) end of checkpoint interval
- : `ckpnt[i].nstep` (`long int`) step counter at ckeckpoint `t0`
- : `ckpnt[i].order` (`int`) method order at checkpoint `t0`
- : `ckpnt[i].step` (`realtype`) step size at checkpoint `t0`

6.2.10 Backward integration of quadrature equations

Not only the backward problem but also the backward quadrature equations may or may not depend on the forward sensitivities. Accordingly, either `CVodeQuadInitB` or `CVodeQuadInitBS` should be used to allocate internal memory and to initialize backward quadratures. For any other operation (extraction, optional input/output, reinitialization, deallocation), the same function is callable regardless of whether or not the quadratures are sensitivity-dependent.

6.2.10.1 Backward quadrature initialization functions

The function `CVodeQuadInitB` initializes and allocates memory for the backward integration of quadrature equations that do not depend on forward sensitivities. It has the following form:

`CVodeQuadInitB`

Call `flag = CVodeQuadInitB(cvode_mem, which, rhsQB, yQB0);`

Description The function `CVodeQuadInitB` provides required problem specifications, allocates internal memory, and initializes backward quadrature integration.

Arguments

`cvode_mem`	(`void *`) pointer to the CVODES memory block.
`which`	(`int`) the identifier of the backward problem.

`rhsQB`	(`CVQuadRhsFnB`) is the C function which computes fQB, the right-hand side of the backward quadrature equations. This function has the form `rhsQB(t, y, yB, qBdot, user_dataB)` (see §6.3.3).
`yQB0`	(`N_Vector`) is the value of the quadrature variables at `tB0`.

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

`CV_SUCCESS`	The call to `CVodeQuadInitB` was successful.
`CV_MEM_NULL`	`cvode_mem` was `NULL`.
`CV_NO_ADJ`	The function `CVodeAdjInit` has not been previously called.
`CV_MEM_FAIL`	A memory allocation request has failed.
`CV_ILL_INPUT`	The parameter `which` is an invalid identifier.

The function `CVodeQuadInitBS` initializes and allocates memory for the backward integration of quadrature equations that depends on the forward sensitivities.

`CVodeQuadInitBS`

Call `flag = CVodeQuadInitBS(cvode_mem, which, rhsQBS, yQBS0);`

Description The function `CVodeQuadInitBS` provides required problem specifications, allocates internal memory, and initializes backward quadrature integration.

Arguments

`cvode_mem`	(`void *`) pointer to the CVODES memory block.
`which`	(`int`) the identifier of the backward problem.
`rhsQBS`	(`CVQuadRhsFnBS`) is the C function which computes $fQBS$, the right-hand side of the backward quadrature equations. This function has the form `rhsQBS(t, y, yS, yB, qBdot, user_dataB)` (see §6.3.4).
`yQBS0`	(`N_Vector`) is the value of the sensitivity-dependent quadrature variables at `tB0`.

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

`CV_SUCCESS`	The call to `CVodeQuadInitBS` was successful.
`CV_MEM_NULL`	`cvode_mem` was `NULL`.
`CV_NO_ADJ`	The function `CVodeAdjInit` has not been previously called.
`CV_MEM_FAIL`	A memory allocation request has failed.
`CV_ILL_INPUT`	The parameter `which` is an invalid identifier.

The integration of quadrature equations during the backward phase can be re-initialized by calling the following function. Before calling `CVodeQuadReInitB` for a new backward problem, call any desired solution extraction functions `CVodeGet**` associated with the previous backward problem.

`CVodeQuadReInitB`

Call `flag = CVodeQuadReInitB(cvode_mem, which, yQB0);`

Description The function `CVodeQuadReInitB` re-initializes the backward quadrature integration.

Arguments

`cvode_mem`	(`void *`) pointer to the CVODES memory block.
`which`	(`int`) the identifier of the backward problem.
`yQB0`	(`N_Vector`) is the value of the quadrature variables at `tB0`.

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

`CV_SUCCESS`	The call to `CVodeQuadReInitB` was successful.
`CV_MEM_NULL`	`cvode_mem` was `NULL`.
`CV_NO_ADJ`	The function `CVodeAdjInit` has not been previously called.
`CV_MEM_FAIL`	A memory allocation request has failed.
`CV_NO_QUAD`	Quadrature integration was not activated through a previous call to `CVodeQuadInitB`.

	CV_ILL_INPUT The parameter which is an invalid identifier.
Notes	The function CVodeQuadReInitB can be called after a call to either CVodeQuadInitB or CVodeQuadInitBS.

6.2.10.2 Backward quadrature extraction function

To extract the values of the quadrature variables at the last return time of CVodeB, CVODES provides a wrapper for the function CVodeGetQuad (see §4.7.3). The call to this function has the form

CVodeGetQuadB

Call	flag = CVodeGetQuadB(cvode_mem, which, &tret, yQB);
Description	The function CVodeGetQuadB returns the quadrature solution vector after a successful return from CVodeB.
Arguments	cvode_mem (void *) pointer to the CVODES memory. tret (realtype) the time reached by the solver (output). yQB (N_Vector) the computed quadrature vector.
Return value	The return value flag of CVodeGetQuadB is one of: CV_SUCCESS CVodeGetQuadB was successful. CV_MEM_NULL cvode_mem is NULL. CV_NO_ADJ The function CVodeAdjInit has not been previously called. CV_NO_QUAD Quadrature integration was not initialized. CV_BAD_DKY yQB was NULL. CV_ILL_INPUT The parameter which is an invalid identifier.

6.2.10.3 Optional input/output functions for backward quadrature integration

Optional values controlling the backward integration of quadrature equations can be changed from their default values through calls to one of the following functions which are wrappers for the corresponding optional input functions defined in §4.7.4. The user must specify the identifier which of the backward problem for which the optional values are specified.

```
flag = CVodeSetQuadErrConB(cvode_mem, which, errconQ);
flag = CVodeQuadSStolerancesB(cvode_mem, which, reltolQ, abstolQ);
flag = CVodeQuadSVtolerancesB(cvode_mem, which, reltolQ, abstolQ);
```

Their return value flag (of type int) can have any of the return values of its counterparts, but it can also be CV_NO_ADJ if the function CVodeAdjInit has not been previously called or CV_ILL_INPUT if the parameter which was an invalid identifier.

Access to optional outputs related to backward quadrature integration can be obtained by calling the corresponding CVodeGetQuad* functions (see §4.7.5). A pointer cvode_memB to the CVODES memory block for the backward problem, required as the first argument of these functions, can be obtained through a call to the functions CVodeGetAdjCVodeBmem (see §6.2.9).

6.3 User-supplied functions for adjoint sensitivity analysis

In addition to the required ODE right-hand side function and any optional functions for the forward problem, when using the adjoint sensitivity module in CVODES, the user must supply one function defining the backward problem ODE and, optionally, functions to supply Jacobian-related information and one or two functions that define the preconditioner (if one of the CVSPILS solvers is selected) for the backward problem. Type definitions for all these user-supplied functions are given below.

6.3.1 ODE right-hand side for the backward problem

If the backward problem does not depend on the forward sensitivities, the user must provide a `rhsB` function of type `CVRhsFnB` defined as follows:

`CVRhsFnB`

Definition
```
typedef int (*CVRhsFnB)(realtype t, N_Vector y,
                        N_Vector yB, N_Vector yBdot, void *user_dataB);
```

Purpose This function evaluates the right-hand side $f_B(t, y, y_B)$ of the backward problem ODE system. This could be either (2.19) or (2.22).

Arguments

`t`	is the current value of the independent variable.
`y`	is the current value of the forward solution vector.
`yB`	is the current value of the backward dependent variable vector.
`yBdot`	is the output vector containing the right-hand side f_B of the backward ODE problem.
`user_dataB`	is a pointer to user data, same as passed to `CVodeSetUserDataB`.

Return value A `CVRhsFnB` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CVodeB` returns `CV_RHSFUNC_FAIL`).

Notes Allocation of memory for `yBdot` is handled within CVODES.

The `y`, `yB`, and `yBdot` arguments are all of type `N_Vector`, but `yB` and `yBdot` typically have different internal representations from `y`. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with CVODES do not perform any consistency checks with respect to their `N_Vector` arguments (see §7.1 and §7.2).

The `user_dataB` pointer is passed to the user's `rhsB` function every time it is called and can be the same as the `user_data` pointer used for the forward problem.

Before calling the user's `rhsB` function, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the right-hand side function which will halt the integration and `CVodeB` will return `CV_RHSFUNC_FAIL`.

6.3.2 ODE right-hand side for the backward problem depending on the forward sensitivities

If the backward problem does depend on the forward sensitivities, the user must provide a `rhsBS` function of type `CVRhsFnBS` defined as follows:

`CVRhsFnBS`

Definition
```
typedef int (*CVRhsFnBS)(realtype t, N_Vector y, N_Vector *yS,
                         N_Vector yB, N_Vector yBdot, void *user_dataB);
```

Purpose This function evaluates the right-hand side $f_B(t, y, y_B, s)$ of the backward problem ODE system. This could be either (2.19) or (2.22).

Arguments

`t`	is the current value of the independent variable.
`y`	is the current value of the forward solution vector.
`yS`	a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution.
`yB`	is the current value of the backward dependent variable vector.

`yBdot` is the output vector containing the right-hand side f_B of the backward ODE problem.

`user_dataB` is a pointer to user data, same as passed to `CVodeSetUserDataB`.

Return value A `CVRhsFnBS` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CVodeB` returns `CV_RHSFUNC_FAIL`).

Notes Allocation of memory for `qBdot` is handled within CVODES.

The `y`, `yB`, and `yBdot` arguments are all of type `N_Vector`, but `yB` and `yBdot` typically have different internal representations from `y`. Likewise for each `yS[i]`. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with CVODES do not perform any consistency checks with respect to their `N_Vector` arguments (see §7.1 and §7.2).

The `user_dataB` pointer is passed to the user's `rhsBS` function every time it is called and can be the same as the `user_data` pointer used for the forward problem.

Before calling the user's `rhsBS` function, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the right-hand side function which will halt the integration and `CVodeB` will return `CV_RHSFUNC_FAIL`.

6.3.3 Quadrature right-hand side for the backward problem

The user must provide an `fQB` function of type `CVQuadRhsFnB` defined by

`CVQuadRhsFnB`

Definition
```
typedef int (*CVQuadRhsFnB)(realtype t, N_Vector y, N_Vector yB,
                            N_Vector qBdot, void *user_dataB);
```

Purpose This function computes the quadrature equation right-hand side for the backward problem.

Arguments `t` is the current value of the independent variable.

`y` is the current value of the forward solution vector.

`yB` is the current value of the backward dependent variable vector.

`qBdot` is the output vector containing the right-hand side `fQB` of the backward quadrature equations.

`user_dataB` is a pointer to user data, same as passed to `CVodeSetUserDataB`.

Return value A `CVQuadRhsFnB` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CVodeB` returns `CV_QRHSFUNC_FAIL`).

Notes Allocation of memory for `rhsvalBQ` is handled within CVODES.

The `y`, `yB`, and `qBdot` arguments are all of type `N_Vector`, but they typically do not all have the same representation. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with CVODES do not perform any consistency checks with repsect to their `N_Vector` arguments (see §7.1 and §7.2).

The `user_dataB` pointer is passed to the user's `fQB` function every time it is called and can be the same as the `user_data` pointer used for the forward problem.

Before calling the user's fQB function, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the quadrature right-hand side function which will halt the integration and CVodeB will return CV_QRHSFUNC_FAIL.

6.3.4 Sensitivity-dependent quadrature right-hand side for the backward problem

The user must provide an fQBS function of type CVQuadRhsFnBS defined by

CVQuadRhsFnBS

Definition
```
typedef int (*CVQuadRhsFnBS)(realtype t, N_Vector y, N_Vector *yS,
                             N_Vector yB, N_Vector qBdot,
                             void *user_dataB);
```

Purpose This function computes the quadrature equation right-hand side for the backward problem.

Arguments

t	is the current value of the independent variable.
y	is the current value of the forward solution vector.
yS	a pointer to an array of Ns vectors containing the sensitivities of the forward solution.
yB	is the current value of the backward dependent variable vector.
qBdot	is the output vector containing the right-hand side fQBS of the backward quadrature equations.
user_dataB	is a pointer to user data, same as passed to CVodeSetUserDataB.

Return value A CVQuadRhsFnBS should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and CVodeB returns CV_QRHSFUNC_FAIL).

Notes Allocation of memory for qBdot is handled within CVODES.

The y, yS, and qBdot arguments are all of type N_Vector, but they typically do not all have the same internal representation. Likewise for each yS[i]. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with CVODES do not perform any consistency checks with repsect to their N_Vector arguments (see §7.1 and §7.2).

The user_dataB pointer is passed to the user's fQBS function every time it is called and can be the same as the user_data pointer used for the forward problem.

Before calling the user's fQBS function, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the quadrature right-hand side function which will halt the integration and CVodeB will return CV_QRHSFUNC_FAIL.

6.3.5 Jacobian information for the backward problem (direct method Jacobian)

If the direct linear solver interface is used for the backward problem (i.e. CVDlsSetLinearSolverB is called in the step described in §6.1), the user may provide a function of type CVDlsJacFnB or CVDlsJacFnBS (see §6.2.8), defined as follows:

CVDlsJacFnB

Definition
```
typedef int (*CVDlsJacFnB)(realtype t, N_Vector y,
                           N_Vector yB, N_Vector fyB,
                           SUNMatrix JacB, void *user_dataB,
                           N_Vector tmp1B, N_Vector tmp2B,
                           N_Vector tmp3B);
```

Purpose This function computes the Jacobian of the backward problem (or an approximation to it).

Arguments

t	is the current value of the independent variable.
y	is the current value of the forward solution vector.
yB	is the current value of the backward dependent variable vector.
fyB	is the current value of the backward right-hand side function f_B.
JacB	is the output approximate Jacobian matrix.
user_dataB	is a pointer to user data – the same as passed to CVodeSetUserDataB.
tmp1B	
tmp2B	
tmp3B	are pointers to memory allocated for variables of type N_Vector which can be used by the CVDlsJacFnB function as temporary storage or work space.

Return value A CVDlsJacFnB should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct, while CVDLS sets last_flag to CVDLS_JACFUNC_RECVR), or a negative value if it failed unrecoverably (in which case the integration is halted, CVodeB returns CV_LSETUP_FAIL and CVDLS sets last_flag to CVDLS_JACFUNC_UNRECVR).

Notes A user-supplied Jacobian function must load the matrix JacB with an approximation to the Jacobian matrix at the point (t,y,yB), where y is the solution of the original IVP at time tt, and yB is the solution of the backward problem at the same time. Information regarding the structure of the specific SUNMATRIX structure (e.g. number of rows, upper/lower bandwidth, sparsity type) may be obtained through using the implementation-specific SUNMATRIX interface functions (see Chapter 8 for details). Only nonzero elements need to be loaded into JacB as this matrix is set to zero before the call to the Jacobian function.

Before calling the user's CVDlsJacFnB, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the Jacobian function which will halt the integration (CVodeB returns CV_LSETUP_FAIL and CVDLS sets last_flag to CVDLS_JACFUNC_UNRECVR).

CVDlsJacFnBS

Definition
```
typedef int (*CVDlsJacFnBS)(realtype t, N_Vector y,
                            N_Vector *yS, N_Vector yB, N_Vector fyB,
                            SUNMatrix JacB, void *user_dataB,
                            N_Vector tmp1B, N_Vector tmp2B,
                            N_Vector tmp3B);
```

Purpose This function computes the Jacobian of the backward problem (or an approximation to it), in the case where the backward problem depends on the forward sensitivities.

Arguments

t	is the current value of the independent variable.
y	is the current value of the forward solution vector.
yS	a pointer to an array of Ns vectors containing the sensitivities of the forward solution.

yB	is the current value of the backward dependent variable vector.
fyB	is the current value of the backward right-hand side function f_B.
JacB	is the output approximate Jacobian matrix.
user_dataB	is a pointer to user data – the same as passed to CVodeSetUserDataB.
tmp1B	
tmp2B	
tmp3B	are pointers to memory allocated for variables of type N_Vector which can be used by CVDlsJacFnBS as temporary storage or work space.

Return value A CVDlsJacFnBS should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct, while CVDLS sets last_flag to CVDLS_JACFUNC_RECVR), or a negative value if it failed unrecoverably (in which case the integration is halted, CVodeB returns CV_LSETUP_FAIL and CVDLS sets last_flag to CVDLS_JACFUNC_UNRECVR).

Notes A user-supplied Jacobian function must load the matrix JacB with an approximation to the Jacobian matrix at the point (t,y,yS,yB), where y is the solution of the original IVP at time tt, yS is the vector of forward sensitivities at time tt, and yB is the solution of the backward problem at the same time. Information regarding the structure of the specific SUNMATRIX structure (e.g. number of rows, upper/lower bandwidth, sparsity type) may be obtained through using the implementation-specific SUNMATRIX interface functions (see Chapter 8 for details). Only nonzero elements need to be loaded into JacB as this matrix is set to zero before the call to the Jacobian function.

Before calling the user's CVDlsDenseJacFnBS, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the Jacobian function which will halt the integration (CVodeB returns CV_LSETUP_FAIL and CVDLS sets last_flag to CVDLS_JACFUNC_UNRECVR).

6.3.6 Jacobian information for the backward problem (matrix-vector product)

If the CVSPILS solver interface is selected for the backward problem (i.e., CVSpilsSetLinearSolverB is called in the steps described in §6.1), the user may provide a function of type CVSpilsJacTimesVecFnB or CVSpilsJacTimesVecFnBS 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.

CVSpilsJacTimesVecFnB

Definition
```
typedef int (*CVSpilsJacTimesVecFnB)(N_Vector vB, N_Vector JvB,
                                     realtype t, N_Vector y, N_Vector yB,
                                     N_Vector fyB, void *user_dataB,
                                     N_Vector tmpB);
```

Purpose This function computes the action of the Jacobian JB for the backward problem on a given vector vB.

Arguments		
	vB	is the vector by which the Jacobian must be multiplied to the right.
	JvB	is the computed output vector JB*vB.
	t	is the current value of the independent variable.
	y	is the current value of the forward solution vector.
	yB	is the current value of the backward dependent variable vector.
	fyB	is the current value of the backward right-hand side function f_B.
	user_dataB	is a pointer to user data – the same as passed to CVodeSetUserDataB.

	tmpB	is a pointer to memory allocated for a variable of type N_Vector which can be used by CVSpilsJacTimesVecFn as temporary storage or work space.

Return value The return value of a function of type CVSpilsJtimesVecFnB should be 0 if successful or nonzero if an error was encountered, in which case the integration is halted.

Notes A user-supplied Jacobian-vector product function must load the vector JvB with the product of the Jacobian of the backward problem at the point (t,y, yB) and the vector vB. Here, y is the solution of the original IVP at time t and yB is the solution of the backward problem at the same time. The rest of the arguments are equivalent to those passed to a function of type CVSpilsJacTimesVecFn (see §4.6.6). If the backward problem is the adjoint of $\dot{y} = f(t, y)$, then this function is to compute $-(\partial f/\partial y)^T v_B$.

CVSpilsJacTimesVecFnBS

Definition
```
typedef int (*CVSpilsJacTimesVecFnBS)(N_Vector vB, N_Vector JvB,
                                      realtype t, N_Vector y, N_Vector *yS,
                                      N_Vector yB, N_Vector fyB,
                                      void *user_dataB, N_Vector tmpB);
```

Purpose This function computes the action of the Jacobian JB for the backward problem on a given vector vB, in the case where the backward problem depends on the forward sensitivities.

Arguments		
	vB	is the vector by which the Jacobian must be multiplied to the right.
	JvB	is the computed output vector JB*vB.
	t	is the current value of the independent variable.
	y	is the current value of the forward solution vector.
	yS	is a pointer to an array containing the forward sensitivity vectors.
	yB	is the current value of the backward dependent variable vector.
	fyB	is the current value of the backward right-hand side function f_B.
	user_dataB	is a pointer to user data – the same as passed to CVodeSetUserDataB.
	tmpB	is a pointer to memory allocated for a variable of type N_Vector which can be used by CVSpilsJacTimesVecFn as temporary storage or work space.

Return value The return value of a function of type CVSpilsJtimesVecFnBS should be 0 if successful or nonzero if an error was encountered, in which case the integration is halted.

Notes A user-supplied Jacobian-vector product function must load the vector JvB with the product of the Jacobian of the backward problem at the point (t,y, yB) and the vector vB. Here, y is the solution of the original IVP at time t and yB is the solution of the backward problem at the same time. The rest of the arguments are equivalent to those passed to a function of type CVSpilsJacTimesVecFn (see §4.6.6).

6.3.7 Jacobian information for the backward problem (matrix-vector setup)

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

CVSpilsJacTimesSetupFnB

Definition
```
typedef int (*CVSpilsJacTimesSetupFnB)(realtype t,
                                       N_Vector y, N_Vector yB,
                                       N_Vector fyB, void *user_dataB);
```

Purpose This function preprocesses and/or evaluates Jacobian data needed by the Jacobian-times-vector routine for the backward problem.

Arguments
- `t` is the current value of the independent variable.
- `y` is the current value of the dependent variable vector, $y(t)$.
- `yB` is the current value of the backward dependent variable vector.
- `fyB` is the current value of the right-hand-side for the backward problem.
- `user_dataB` is a pointer to user data — the same as the `user_dataB` parameter passed to `CVSetUserDataB`.

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

Notes Each call to the Jacobian-vector setup function is preceded by a call to the backward problem residual user function with the same `(t,y, yB)` arguments. Thus, the setup function can use any auxiliary data that is computed and saved during the evaluation of the right-hand-side function.

If the user's `CVSpilsJacTimesVecFnB` 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 `cvode_mem` to `user_dataB` and then use the `CVGet*` functions described in §4.5.8.2. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

`CVSpilsJacTimesSetupFnBS`

Definition
```
typedef int (*CVSpilsJacTimesSetupFnBS)(realtype t,
                                        N_Vector y, N_Vector *yS,
                                        N_Vector yB, N_Vector fyB,
                                        void *user_dataB);
```

Purpose This function preprocesses and/or evaluates Jacobian data needed by the Jacobian-times-vector routine for the backward problem, in the case that the backward problem depends on the forward sensitivities.

Arguments
- `t` is the current value of the independent variable.
- `y` is the current value of the dependent variable vector, $y(t)$.
- `yS` a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution.
- `yB` is the current value of the backward dependent variable vector.
- `fyB` is the current value of the right-hand-side function for the backward problem.
- `user_dataB` is a pointer to user data — the same as the `user_dataB` parameter passed to `CVSetUserDataB`.

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

Notes Each call to the Jacobian-vector setup function is preceded by a call to the backward problem residual user function with the same `(t,y, yS, yB)` arguments. Thus, the setup function can use any auxiliary data that is computed and saved during the evaluation of the right-hand-side function.

If the user's `CVSpilsJacTimesVecFnB` 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 `cvode_mem` to `user_dataB` and then use the `CVGet*` functions described in §4.5.8.2. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

6.3.8 Preconditioning for the backward problem (linear system solution)

If preconditioning is used during integration of the backward problem, then the user must provide a C function to solve the linear system $Pz = r$, where P may be either a left or a right preconditioner matrix. Here P should approximate (at least crudely) the Newton matrix $M_B = I - \gamma_B J_B$, where $J_B = \partial f_B / \partial y_B$. If preconditioning is done on both sides, the product of the two preconditioner matrices should approximate M_B. This function must be of one of the following two types:

`CVSpilsPrecSolveFnB`

Definition
```
typedef int (*CVSpilsPrecSolveFnB)(realtype t, N_Vector y,
                                   N_Vector yB, N_Vector fyB,
                                   N_Vector rvecB, N_Vector zvecB,
                                   realtype gammaB, realtype deltaB,
                                   void *user_dataB);
```

Purpose This function solves the preconditioning system $Pz = r$ for the backward problem.

Arguments

`t`	is the current value of the independent variable.
`y`	is the current value of the forward solution vector.
`yB`	is the current value of the backward dependent variable vector.
`fyB`	is the current value of the backward right-hand side function f_B.
`rvecB`	is the right-hand side vector r of the linear system to be solved.
`zvecB`	is the computed output vector.
`gammaB`	is the scalar appearing in the Newton matrix, $M_B = I - \gamma_B J_B$.
`deltaB`	is an input tolerance to be used if an iterative method is employed in the solution.
`user_dataB`	is a pointer to user data — the same as the `user_dataB` parameter passed to `CVodeSetUserDataB`.

Return value The return value of a preconditioner solve function for the backward problem 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).

`CVSpilsPrecSolveFnBS`

Definition
```
typedef int (*CVSpilsPrecSolveFnBS)(realtype t, N_Vector y, N_Vector *yS,
                                    N_Vector yB, N_Vector fyB,
                                    N_Vector rvecB, N_Vector zvecB,
                                    realtype gammaB, realtype deltaB,
                                    void *user_dataB);
```

Purpose This function solves the preconditioning system $Pz = r$ for the backward problem, in the case where the backward problem depends on the forward sensitivities.

Arguments

`t`	is the current value of the independent variable.
`y`	is the current value of the forward solution vector.
`yS`	is a pointer to an array containing the forward sensitivity vectors.
`yB`	is the current value of the backward dependent variable vector.
`fyB`	is the current value of the backward right-hand side function f_B.
`rvecB`	is the right-hand side vector r of the linear system to be solved.
`zvecB`	is the computed output vector.
`gammaB`	is the scalar appearing in the Newton matrix, $M_B = I - \gamma_B J_B$.
`deltaB`	is an input tolerance to be used if an iterative method is employed in the solution.

`user_dataB` is a pointer to user data — the same as the `user_dataB` parameter passed to `CVodeSetUserDataB`.

Return value The return value of a preconditioner solve function for the backward problem 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).

6.3.9 Preconditioning for the backward problem (Jacobian data)

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 C function of one of the following two types:

`CVSpilsPrecSetupFnB`

Definition
```
typedef int (*CVSpilsPrecSetupFnB)(realtype t, N_Vector y,
                                   N_Vector yB, N_Vector fyB,
                                   booleantype jokB, booleantype *jcurPtrB,
                                   realtype gammaB, void *user_dataB);
```

Purpose This function preprocesses and/or evaluates Jacobian-related data needed by the preconditioner for the backward problem.

Arguments The arguments of a `CVSpilsPrecSetupFnB` are as follows:

`t`	is the current value of the independent variable.
`y`	is the current value of the forward solution vector.
`yB`	is the current value of the backward dependent variable vector.
`fyB`	is the current value of the backward right-hand side function f_B.
`jokB`	is an input flag indicating whether Jacobian-related data needs to be recomputed (`jokB`=`SUNFALSE`) or information saved from a previous invokation can be safely used (`jokB`=`SUNTRUE`).
`jcurPtr`	is an output flag which must be set to `SUNTRUE` if Jacobian-relatd data was recomputed or `SUNFALSE` otherwise.
`gammaB`	is the scalar appearing in the Newton matrix.
`user_dataB`	is a pointer to user data — the same as the `user_dataB` parameter passed to `CVodeSetUserDataB`.

Return value The return value of a preconditioner setup function for the backward problem 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).

`CVSpilsPrecSetupFnBS`

Definition
```
typedef int (*CVSpilsPrecSetupFnBS)(realtype t, N_Vector y, N_Vector *yS,
                                    N_Vector yB, N_Vector fyB,
                                    booleantype jokB, booleantype *jcurPtrB,
                                    realtype gammaB, void *user_dataB);
```

Purpose This function preprocesses and/or evaluates Jacobian-related data needed by the preconditioner for the backward problem, in the case where the backward problem depends on the forward sensitivities.

Arguments The arguments of a `CVSpilsPrecSetupFnBS` are as follows:

`t`	is the current value of the independent variable.
`y`	is the current value of the forward solution vector.
`yS`	is a pointer to an array containing the forward sensitivity vectors.
`yB`	is the current value of the backward dependent variable vector.

`fyB`	is the current value of the backward right-hand side function f_B.
`jokB`	is an input flag indicating whether Jacobian-related data needs to be recomputed (`jokB=SUNFALSE`) or information saved from a previous invokation can be safely used (`jokB=SUNTRUE`).
`jcurPtr`	is an output flag which must be set to `SUNTRUE` if Jacobian-relatd data was recomputed or `SUNFALSE` otherwise.
`gammaB`	is the scalar appearing in the Newton matrix.
`user_dataB`	is a pointer to user data — the same as the `user_dataB` parameter passed to `CVodeSetUserDataB`.

Return value The return value of a preconditioner setup function for the backward problem 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).

6.4 Using CVODES preconditioner modules for the backward problem

As on the forward integration phase, the efficiency of Krylov iterative methods for the solution of linear systems can be greatly enhanced through preconditioning. Both preconditioner modules provided with SUNDIALS, the serial banded preconditioner CVBANDPRE and the parallel band-block-diagonal preconditioner module CVBBDPRE, provide interface functions through which they can be used on the backward integration phase.

6.4.1 Using the banded preconditioner CVBANDPRE

The adjoint module in CVODES offers an interface to the banded preconditioner module CVBANDPRE described in section §4.8.1. This preconditioner, usable only in a serial setting, provides a band matrix preconditioner based on difference quotients of the backward problem right-hand side function `fB`. It generates a banded approximation to the Jacobian with m_{lB} sub-diagonals and m_{uB} super-diagonals to be used with one of the Krylov linear solvers.

In order to use the CVBANDPRE module in the solution of the backward problem, the user need not define any additional functions. Instead, *after* one of the CVSPILS linear solvers has been specified, by calling the appropriate function (see §6.2.5), the following call to the CVBANDPRE module initialization function must be made.

`CVBandPrecInitB`

Call `flag = CVBandPrecInitB(cvode_mem, which, nB, muB, mlB);`

Description The function `CVBandPrecInitB` initializes and allocates memory for the CVBANDPRE preconditioner for the backward problem. It creates, allocates, and stores (internally in the CVODES solver block) a pointer to the newly created CVBANDPRE memory block.

Arguments

`cvode_mem`	(`void *`) pointer to the CVODES memory block.
`which`	(`int`) the identifier of the backward problem.
`nB`	(`sunindextype`) backward problem dimension.
`muB`	(`sunindextype`) upper half-bandwidth of the backward problem Jacobian approximation.
`mlB`	(`sunindextype`) lower half-bandwidth of the backward problem Jacobian approximation.

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

`CVSPILS_SUCCESS`	The call to `CVodeBandPrecInitB` was successful.
`CVSPILS_MEM_FAIL`	A memory allocation request has failed.

CVSPILS_MEM_NULL	The cvode_mem argument was NULL.
CVSPILS_LMEM_NULL	No linear solver has been attached.
CVSPILS_ILL_INPUT	An invalid parameter has been passed.

For more details on CVBANDPRE see §4.8.1.

6.4.2 Using the band-block-diagonal preconditioner CVBBDPRE

The adjoint module in CVODES offers an interface to the band-block-diagonal preconditioner module CVBBDPRE described in section §4.8.2. This generates a preconditioner that is a block-diagonal matrix with each block being a band matrix and can be used with one of the Krylov linear solvers and with the MPI-parallel vector module NVECTOR_PARALLEL.

In order to use the CVBBDPRE module in the solution of the backward problem, the user must define one or two additional functions, described at the end of this section.

6.4.2.1 Initialization of CVBBDPRE

The CVBBDPRE module is initialized by calling the following function, *after* one of the CVSPILS linear solvers has been specified by calling the appropriate function (see §6.2.5).

CVBBDPrecInitB

Call
```
flag = CVBBDPrecInitB(cvode_mem, which, NlocalB, mudqB, mldqB,
                      mukeepB, mlkeepB, dqrelyB, glocB, gcommB);
```

Description The function CVBBDPrecInitB initializes and allocates memory for the CVBBDPRE preconditioner for the backward problem. It creates, allocates, and stores (internally in the CVODES solver block) a pointer to the newly created CVBBDPRE memory block.

Arguments

cvode_mem	(void *) pointer to the CVODES memory block.
which	(int) the identifier of the backward problem.
NlocalB	(sunindextype) local vector dimension for the backward problem.
mudqB	(sunindextype) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
mldqB	(sunindextype) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
mukeepB	(sunindextype) upper half-bandwidth of the retained banded approximate Jacobian block.
mlkeepB	(sunindextype) lower half-bandwidth of the retained banded approximate Jacobian block.
dqrelyB	(realtype) the relative increment in components of yB used in the difference quotient approximations. The default is dqrelyB= $\sqrt{\text{unit roundoff}}$, which can be specified by passing dqrely= 0.0.
glocB	(CVBBDLocalFnB) the C function which computes the function $g_B(t, y, y_B)$ approximating the right-hand side of the backward problem.
gcommB	(CVBBDCommFnB) the optional C function which performs all interprocess communication required for the computation of g_B.

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

CVSPILS_SUCCESS	The call to CVodeBBDPrecInitB was successful.
CVSPILS_MEM_FAIL	A memory allocation request has failed.
CVSPILS_MEM_NULL	The cvode_mem argument was NULL.
CVSPILS_LMEM_NULL	No linear solver has been attached.
CVSPILS_ILL_INPUT	An invalid parameter has been passed.

To reinitialize the CVBBDPRE preconditioner module for the backward problem, possibly with changes in mudqB, mldqB, or dqrelyB, call the following function:

CVBBDPrecReInitB

Call `flag = CVBBDPrecReInitB(cvode_mem, which, mudqB, mldqB, dqrelyB);`

Description The function `CVBBDPrecReInitB` reinitializes the CVBBDPRE preconditioner for the backward problem.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block returned by `CVodeCreate`.
- `which` (`int`) the identifier of the backward problem.
- `mudqB` (`sunindextype`) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
- `mldqB` (`sunindextype`) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
- `dqrelyB` (`realtype`) the relative increment in components of `yB` used in the difference quotient approximations.

Return value The return value `flag` (of type `int`) is one of:
- `CVSPILS_SUCCESS` The call to `CVodeBBDPrecReInitB` was successful.
- `CVSPILS_MEM_FAIL` A memory allocation request has failed.
- `CVSPILS_MEM_NULL` The `cvode_mem` argument was `NULL`.
- `CVSPILS_PMEM_NULL` The `CVodeBBDPrecInitB` has not been previously called.
- `CVSPILS_LMEM_NULL` No linear solver has been attached.
- `CVSPILS_ILL_INPUT` An invalid parameter has been passed.

For more details on CVBBDPRE see §4.8.2.

6.4.2.2 User-supplied functions for CVBBDPRE

To use the CVBBDPRE module, the user must supply one or two functions which the module calls to construct the preconditioner: a required function `glocB` (of type `CVBBDLocalFnB`) which approximates the right-hand side of the backward problem and which is computed locally, and an optional function `gcommB` (of type `CVBBDCommFnB`) which performs all interprocess communication necessary to evaluate this approximate right-hand side (see §4.8.2). The prototypes for these two functions are described below.

CVBBDLocalFnB

Definition
```
typedef int (*CVBBDLocalFnB)(sunindextype NlocalB, realtype t, N_Vector y,
                             N_Vector yB, N_Vector gB, void *user_dataB);
```

Purpose This `glocB` function loads the vector `gB`, an approximation to the right-hand side f_B of the backward problem, as a function of `t`, `y`, and `yB`.

Arguments
- `NlocalB` is the local vector length for the backward problem.
- `t` is the value of the independent variable.
- `y` is the current value of the forward solution vector.
- `yB` is the current value of the backward dependent variable vector.
- `gB` is the output vector, $g_B(t, y, y_B)$.
- `user_dataB` is a pointer to user data — the same as the `user_dataB` parameter passed to `CVodeSetUserDataB`.

Return value An `CVBBDLocalFnB` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CVodeB` returns `CV_LSETUP_FAIL`).

Notes This routine must assume that all interprocess communication of data needed to calculate `gB` has already been done, and this data is accessible within `user_dataB`.

Before calling the user's CVBBDLocalFnB, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the preconditioner setup function which will halt the integration (CVodeB returns CV_LSETUP_FAIL).

CVBBDCommFnB

Definition
```
typedef int (*CVBBDCommFnB)(sunindextype NlocalB, realtype t, N_Vector y,
                            N_Vector yB, void *user_dataB);
```

Purpose This gcommB function must perform all interprocess communications necessary for the execution of the glocB function above, using the input vectors y and yB.

Arguments
- NlocalB is the local vector length.
- t is the value of the independent variable.
- y is the current value of the forward solution vector.
- yB is the current value of the backward dependent variable vector.
- user_dataB is a pointer to user data — the same as the user_dataB parameter passed to CVodeSetUserDataB.

Return value An CVBBDCommFnB should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and CVodeB returns CV_LSETUP_FAIL).

Notes The gcommB function is expected to save communicated data in space defined within the structure user_dataB.

Each call to the gcommB function is preceded by a call to the function that evaluates the right-hand side of the backward problem with the same t, y, and yB, arguments. If there is no additional communication needed, then pass gcommB = NULL to CVBBDPrecInitB.

Chapter 7

Description of the NVECTOR module

The SUNDIALS solvers are written in a data-independent manner. They all operate on generic vectors (of type `N_Vector`) through a set of operations defined by the particular NVECTOR implementation. Users can provide their own specific implementation of the NVECTOR module, or use one of the implementations provided with SUNDIALS. The generic 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    (*nvl1norm)(N_Vector);
  void        (*nvcompare)(realtype, N_Vector, N_Vector);
  booleantype (*nvinvtest)(N_Vector, N_Vector);
  booleantype (*nvconstrmask)(N_Vector, N_Vector, N_Vector);
  realtype    (*nvminquotient)(N_Vector, N_Vector);
};
```

The generic NVECTOR module defines and implements the vector operations acting on `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 7.2 contains a complete list of all vector operations defined by the generic NVECTOR module.

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:

- 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 7.1. It is recommended that a user-supplied NVECTOR implementation use the `SUNDIALS_NVEC_CUSTOM` identifier.

Table 7.1: Vector Identifications associated with vector kernels supplied with SUNDIALS.

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

Table 7.2: Description of the NVECTOR operations

Name	Usage and Description
`N_VGetVectorID`	`id = N_VGetVectorID(w);` Returns the vector type identifier for the vector `w`. It is used to determine the vector implementation type (e.g. serial, parallel,...) from the abstract `N_Vector` interface. Returned values are given in Table 7.1.
`N_VClone`	`v = N_VClone(w);` Creates a new `N_Vector` of the same type as an existing vector `w` and sets the *ops* field. It does not copy the vector, but rather allocates storage for the new vector.
`N_VCloneEmpty`	`v = N_VCloneEmpty(w);` Creates a new `N_Vector` of the same type as an existing vector `w` and sets the *ops* field. It does not allocate storage for data.
`N_VDestroy`	`N_VDestroy(v);` Destroys the `N_Vector v` and frees memory allocated for its internal data.
`N_VSpace`	`N_VSpace(nvSpec, &lrw, &liw);` Returns storage requirements for one `N_Vector`. `lrw` contains the number of realtype words and `liw` contains the number of integer words. This function is advisory only, for use in determining a user's total space requirements; it could be a dummy function in a user-supplied NVECTOR module if that information is not of interest.

continued on next page

continued from last page

Name	Usage and Description
N_VGetArrayPointer	`vdata = N_VGetArrayPointer(v);` Returns a pointer to a `realtype` array from the `N_Vector v`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realtype`. 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.
N_VSetArrayPointer	`N_VSetArrayPointer(vdata, v);` Overwrites the data in an `N_Vector` with a given array of `realtype`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realtype`. This routine is only used in the interfaces to the dense (serial) linear solver, hence need not exist in a user-supplied NVECTOR module for a parallel environment.
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`: $z_i = ax_i + by_i$, $i = 0, \ldots, n-1$.
N_VConst	`N_VConst(c, z);` Sets all components of the `N_Vector z` to `realtype c`: $z_i = c$, $i = 0, \ldots, n-1$.
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`: $z_i = x_i y_i$, $i = 0, \ldots, n-1$.
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`: $z_i = x_i/y_i$, $i = 0, \ldots, n-1$. The y_i may not be tested for 0 values. It should only be called with a `y` that is guaranteed to have all nonzero components.
N_VScale	`N_VScale(c, x, z);` Scales the `N_Vector x` by the `realtype` scalar `c` and returns the result in `z`: $z_i = cx_i$, $i = 0, \ldots, n-1$.
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`: $y_i = \lvert x_i \rvert$, $i = 0, \ldots, n-1$.

continued on next page

continued from last page

Name	Usage and Description
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`: $z_i = 1.0/x_i$, $i = 0, \ldots, n-1$. This routine may not check for division by 0. It should be called only with an `x` which is guaranteed to have all nonzero components.
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`: $z_i = x_i + b$, $i = 0, \ldots, n-1$.
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`: $m = \max_i \lvert x_i \rvert$.
N_VWrmsNorm	`m = N_VWrmsNorm(x, w)` Returns the weighted root-mean-square norm of the `N_Vector x` with `realtype` weight vector `w`: $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i)^2\right)/n}$.
N_VWrmsNormMask	`m = N_VWrmsNormMask(x, w, id);` Returns the weighted root mean square norm of the `N_Vector x` with `realtype` weight vector `w` built using only the elements of `x` corresponding to positive elements of the `N_Vector id`: $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i msk_i)^2\right)/n}$, where msk_i is 1 if $id_i > 0$ or 0 if $id_i \leq 0$.
N_VMin	`m = N_VMin(x);` Returns the smallest element of the `N_Vector x`: $m = \min_i x_i$.
N_VWL2Norm	`m = N_VWL2Norm(x, w);` Returns the weighted Euclidean ℓ_2 norm of the `N_Vector x` with `realtype` weight vector `w`: $m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2}$.
N_VL1Norm	`m = N_VL1Norm(x);` Returns the ℓ_1 norm of the `N_Vector x`: $m = \sum_{i=0}^{n-1} \lvert x_i \rvert$.
N_VCompare	`N_VCompare(c, x, z);` Compares the components of the `N_Vector x` to the `realtype` scalar `c` and returns an `N_Vector z` such that: $z_i = 1.0$ if $\lvert x_i \rvert \geq c$ and $z_i = 0.0$ otherwise.

continued on next page

continued from last page	
Name	**Usage and Description**
N_VInvTest	`t = N_VInvTest(x, z);` Sets the components of the `N_Vector z` to be the inverses of the components of the `N_Vector x`, with prior testing for zero values: $z_i = 1.0/x_i$, $i = 0, \ldots, n-1$. This routine returns a boolean assigned to `SUNTRUE` if all components of `x` are nonzero (successful inversion) and returns `SUNFALSE` otherwise.
N_VConstrMask	`t = N_VConstrMask(c, x, 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 `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.
N_VMinQuotient	`minq = N_VMinQuotient(num, denom);` This routine returns the minimum of the quotients obtained by termwise dividing $\texttt{num}_i$ by $\texttt{denom}_i$. 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.

7.1 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 Table 7.2. Their names are obtained from those in Table 7.2 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.

7.2 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 Table 7.2 Their names are obtained from those in Table 7.2 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.

7.3 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 Table 7.2. Their names are obtained from those in Table 7.2 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 a 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.

7.4 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 Table 7.2. Their names are obtained from those in Table 7.2 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.

7.5 The NVECTOR_PARHYP implementation

The NVECTOR_PARHYP implementation of the NVECTOR module provided with SUNDIALS is a wrapper around *hypre*'s ParVector class. Most of the vector kernels simply call *hypre* vector operations. The implementation defines the *content* field of `N_Vector` to be a structure containing the global and local lengths of the vector, a pointer to an object of type `hypre_ParVector`, an MPI communicator, and a boolean flag *own_parvector* indicating ownership of the *hypre* parallel vector object x.

```
struct _N_VectorContent_ParHyp {
  sunindextype local_length;
  sunindextype global_length;
  booleantype own_parvector;
  MPI_Comm comm;
  hypre_ParVector *x;
};
```

The header file to include when using this module is `nvector_parhyp.h`. The installed module library to link to is `libsundials_nvecparhyp.`*`lib`* where *`.lib`* is typically `.so` for shared libraries and `.a` for static libraries.

Unlike native SUNDIALS vector types, NVECTOR_PARHYP does not provide macros to access its member variables. Note that NVECTOR_PARHYP requires SUNDIALS to be built with MPI support.

The NVECTOR_PARHYP module defines implementations of all vector operations listed in Table 7.2, except for `N_VSetArrayPointer` and `N_VGetArrayPointer`, because accessing raw vector data is handled by low-level *hypre* functions. As such, this vector is not available for use with SUNDIALS Fortran interfaces. When access to raw vector data is needed, one should extract the *hypre* vector first, and then use *hypre* methods to access the data. Usage examples of NVECTOR_PARHYP are provided in the `cvAdvDiff_non_ph.c` example program for CVODE [23] and the `ark_diurnal_kry_ph.c` example program for ARKODE [31].

The names of parhyp methods are obtained from those in Table 7.2 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 *hypre* 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.

7.6 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 Table 7.2, except for `N_VGetArrayPointer` and `N_VSetArrayPointer`. As such, this vector cannot be used with SUNDIALS Fortran interfaces. When access to raw vector data is needed, it is recommended to extract the PETSc vector first, and then use PETSc methods to access the data. Usage examples of NVECTOR_PETSC are provided in example programs for IDA [22].

The names of vector operations are obtained from those in Table 7.2 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 recommeded to extract the PETSc vector via `x_vec = N_VGetVector_Petsc(v)` and then access components using appropriate PETSc functions.
- The functions `N_VNewEmpty_Petsc`, `N_VMake_Petsc`, and `N_VCloneVectorArrayEmpty_Petsc` set the field *own_data* to `SUNFALSE`. `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.

7.7 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 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 `SUNDIALS_Comm`.

The NVECTOR_CUDA module defines implementations of all vector operations listed in Table 7.2, 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 [23].

The names of vector operations are obtained from those in Table 7.2 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 recommeded 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.

7.8 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 `sunrajavec`.

- `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 Table 7.2, 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. 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 [23].

The names of vector operations are obtained from those in Table 7.2 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 recommeded 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.

7.9 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 max norm.
- `Test_N_VWrmsNorm`: Create vector of known values, find and validate weighted root mean square.

- `Test_N_VWrmsNormMask`: Case 1: Create vector of known values, find and validate weighted root mean square using all elements.
- `Test_N_VWrmsNormMask`: Case 2: Create vector of known values, find and validate weighted root mean square using no elements.
- `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.

7.10 NVECTOR functions used by CVODES

In Table 7.3 below, we list the vector functions in the NVECTOR module used within the CVODES package. The table also shows, for each function, which of the code modules uses the function. The CVODES column shows function usage within the main integrator module, while the remaining columns show function usage within each of the CVODES linear solver interfaces, the CVBANDPRE and CVBBDPRE preconditioner modules, and the CVODES adjoint sensitivity module (denoted here by CVODEA). Here CVDLS stands for the direct linear solver interface in CVODES; CVSPILS stands for the scaled, preconditioned, iterative linear solver interface in CVODES.

At this point, we should emphasize that the CVODES user does not need to know anything about the usage of vector functions by the CVODES code modules in order to use CVODES. The information is presented as an implementation detail for the interested reader.

The vector functions listed in Table 7.2 that are *not* used by CVODES are: `N_VWL2Norm`, `N_VL1Norm`, `N_VWrmsNormMask`, and `N_VCloneEmpty`. Therefore, a user-supplied NVECTOR module for CVODES could omit these kernels. 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.

Table 7.3: List of vector functions usage by CVODES code modules

	CVODES	CVDLS	CVDIAG	CVSPILS	CVBANDPRE	CVBBDPRE	CVODEA
N_VGetVectorID							
N_VClone	✓		✓	✓			✓
N_VDestroy	✓		✓	✓			✓
N_VCloneVectorArray	✓						✓
N_VDestroyVectorArray	✓						✓
N_VSpace	✓						
N_VGetArrayPointer		✓			✓	✓	
N_VSetArrayPointer		✓					
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_MinQuotient	✓						
N_VConstrMask	✓						
N_VCompare	✓		✓				
N_VInvTest			✓				

Chapter 8

Description of the SUNMatrix module

For problems that involve direct methods for solving linear systems, the SUNDIALS solvers not only operate on generic vectors, but also on generic matrices (of type `SUNMatrix`), through a set of operations defined by the particular SUNMATRIX implementation. Users can provide their own specific implementation of the SUNMATRIX module, particularly in cases where they provide their own NVECTOR and/or linear solver modules, and require matrices that are compatible with those implementations. Alternately, we provide three SUNMATRIX implementations: dense, banded, and sparse. The generic operations are described below, and descriptions of the implementations provided with SUNDIALS follow.

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 8.1: 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 8.2 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 8.1. It is recommended that a user-supplied SUNMATRIX implementation use the `SUNMATRIX_CUSTOM` identifier.

Table 8.2: Description of the `SUNMatrix` operations

Name	Usage and Description
`SUNMatGetID`	`id = SUNMatGetID(A);` Returns the type identifier for the matrix `A`. It is used to determine the matrix implementation type (e.g. dense, banded, sparse,...) from the abstract `SUNMatrix` interface. This is used to assess compatibility with SUNDIALS-provided linear solver implementations. Returned values are given in the Table 8.1.

continued on next page

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

We note that not all SUNMATRIX types are compatible with all NVECTOR types provided with SUNDIALS. This is primarily due to the need for compatibility within the `SUNMatMatvec` routine; however, compatibility between SUNMATRIX and NVECTOR implementations is more crucial when considering their interaction within SUNLINSOL objects, as will be described in more detail in Chapter 9. More specifically, in Table 8.3 we show the matrix interfaces available as SUNMATRIX modules, and the compatible vector implementations.

Table 8.3: SUNDIALS matrix interfaces and vector implementations that can be used for each.

Matrix Interface	Serial	Parallel (MPI)	OpenMP	pThreads	*hypre* Vec.	PETSc Vec.	CUDA	RAJA	User Suppl.
Dense	✓		✓	✓					✓

continued on next page

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

8.1 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 $0 \le$ `i` $<$ `M` and $0 \le$ `j` $<$ `N`) may be accessed via `data[j*M+i]`.

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

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

The header file to include when using this module is `sunmatrix/sunmatrix_dense.h`. The 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_LDATA_D`

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

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

Implementation:

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

- `SM_DATA_D` and `SM_COLS_D`

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

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

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

  Implementation:

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

- `SM_COLUMN_D` and `SM_ELEMENT_D`

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

  The assignment `col_j = SM_COLUMN_D(A,j)` sets `col_j` to be a pointer to the first entry of the `j`-th column of the $\mathtt{M} \times \mathtt{N}$ dense matrix `A` (with $0 \le \mathtt{j} < \mathtt{N}$). 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 $\mathtt{M} - 1$.

  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 $\mathtt{M} \times \mathtt{N}$ dense matrix `A` (with $0 \le \mathtt{i} < \mathtt{M}$ and $0 \le \mathtt{j} < \mathtt{N}$).

  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 8.2. Their names are obtained from those in Table 8.2 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 jth column of the dense SUNMatrix. The resulting pointer should be indexed over the range 0 to $\texttt{M} - 1$.

  ```
  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.

8.2 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 8.1. 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 ≤ `mu` < `N`

ml - lower half-bandwidth, 0 ≤ `ml` < `N`

s_mu - storage upper bandwidth, `mu` ≤ `s_mu` < `N`. The LU decomposition routines in the associated SUNLINSOL_BAND and SUNLINSOL_LAPACKBAND modules write the LU factors into the storage for A. The upper triangular factor U, however, may have an upper bandwidth as big as min(`N`-1,`mu`+`ml`) because of partial pivoting. The `s_mu` field holds the upper half-bandwidth allocated for A.

ldim - leading dimension (`ldim` ≥ `s_mu`+`ml`+1)

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

ldata - length of the data array (= `ldim`·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` ≤ i ≤ 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.

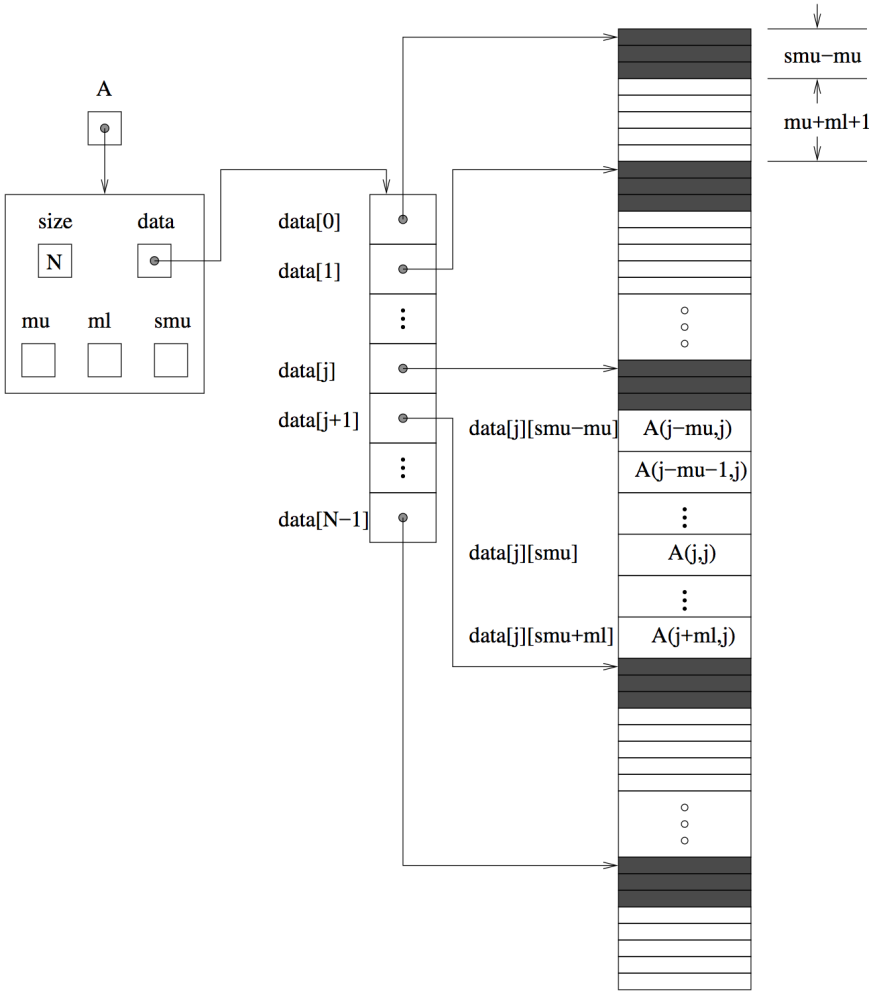


Figure 8.1: Diagram of the storage for the SUNMATRIX_BAND module. Here `A` is an `N` × `N` band matrix with upper and lower half-bandwidths `mu` and `ml`, respectively. The rows and columns of `A` are numbered from 0 to `N` − 1 and the (i, j)-th element of `A` is denoted `A(i,j)`. The greyed out areas of the underlying component storage are used by the associated SUNLINSOL_BAND linear solver.

- SM_CONTENT_B

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

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

  Implementation:

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

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

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

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

  Implementation:

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

- SM_DATA_B and SM_COLS_B

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

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

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

  Implementation:

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

- SM_COLUMN_B, SM_COLUMN_ELEMENT_B, and SM_ELEMENT_B

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

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

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

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

Implementation:

```
#define SM_COLUMN_B(A,j)    ( ((SM_CONTENT_B(A)->cols)[j])+SM_SUBAND_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_SUBAND_B(A)] )
```

The SUNMATRIX_BAND module defines banded implementations of all matrix operations listed in Table 8.2. Their names are obtained from those in Table 8.2 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 `mu` 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 ml.

  ```
  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.

8.3 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 8.2 (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) )
  ```

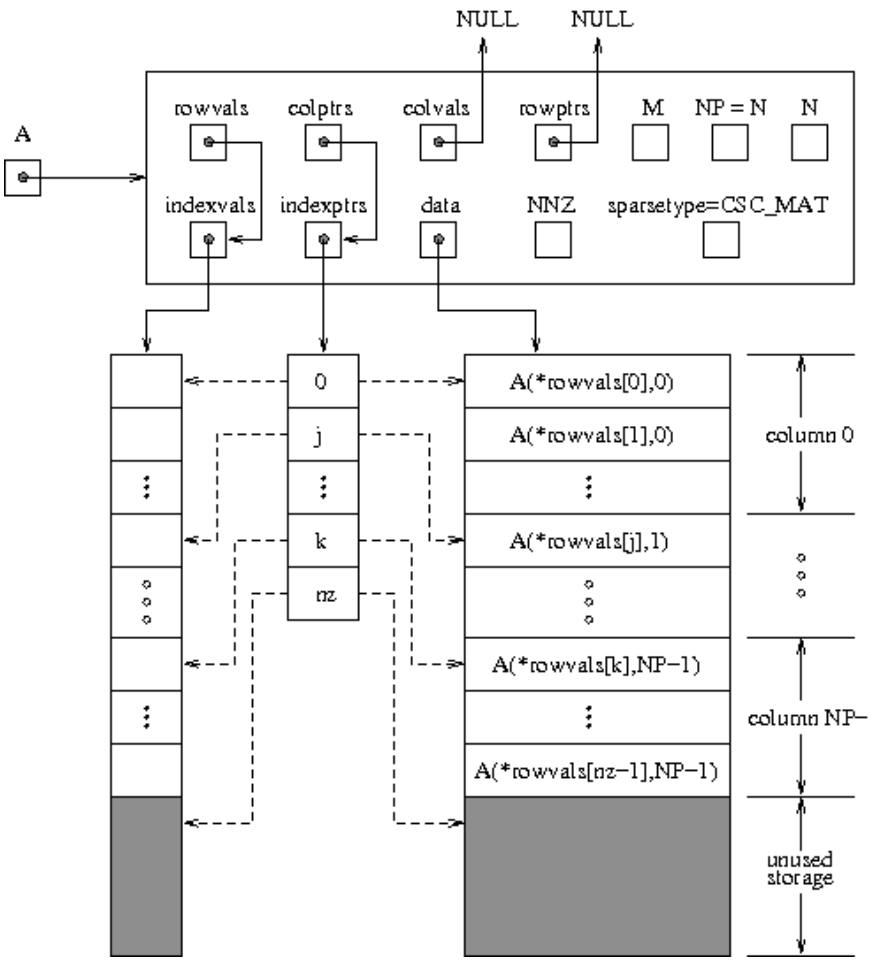


Figure 8.2: Diagram of the storage for a compressed-sparse-column matrix. Here `A` is an $\mathtt{M} \times \mathtt{N}$ sparse matrix with storage for up to `NNZ` nonzero entries (the allocated length of both `data` and `indexvals`). The entries in `indexvals` may assume values from 0 to $\mathtt{M}-1$, 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 $\mathtt{N}+1$ 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 8.2. Their names are obtained from those in Table 8.2 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.

8.4 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`.

8.5 SUNMatrix functions used by CVODES

In Table 8.4 below, we list the matrix functions in the SUNMATRIX module used within the CVODES package. The table also shows, for each function, which of the code modules uses the function. Neither the main CVODES integrator or the CVSPILS interface call SUNMATRIX functions directly, so the table columns are specific to the CVDLS direct solver interface and the CVBANDPRE and CVBBDPRE preconditioner modules.

At this point, we should emphasize that the CVODES user does not need to know anything about the usage of matrix functions by the CVODES code modules in order to use CVODES. The information is presented as an implementation detail for the interested reader.

Table 8.4: List of matrix functions usage by CVODES code modules

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

The matrix functions listed in Table 8.2 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 8.2 that are *not* used by CVODES are: `SUNMatScaleAdd` and `SUNMatMatvec`. Therefore a user-supplied SUNMATRIX module for CVODES could omit these functions.

Chapter 9

Description of the SUNLinearSolver module

For problems that involve the solution of linear systems of equations, the SUNDIALS solvers operate using generic linear solver modules (of type `SUNLinearSolver`), through a set of operations defined by the particular SUNLINSOL implementation. These work in coordination with the SUNDIALS generic NVECTOR and SUNMATRIX modules to provide a set of compatible data structures and solvers for the solution of linear systems using direct or iterative methods. Moreover, users can provide their own specific SUNLINSOL implementation to each SUNDIALS solver, particularly in cases where they provide their own NVECTOR and/or SUNMATRIX modules, and the customized linear solver leverages these additional data structures to create highly efficient and/or scalable solvers for their particular problem. Additionally, SUNDIALS provides native implementations SUNLINSOL modules, as well as SUNLINSOL modules that interface between SUNDIALS and external linear solver libraries.

The various SUNDIALS solvers have been designed to specifically leverage the use of either *direct linear solvers* or *scaled, preconditioned, iterative linear solvers*, through their "Dls" and "Spils" interfaces, respectively. Additionally, SUNDIALS solvers can make use of user-supplied custom linear solvers, whether these are problem-specific or come from external solver libraries.

For iterative (and possibly custom) linear solvers, the SUNDIALS solvers 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, we apply the underlying iterative algorithm to the transformed system

$$\tilde{A}\tilde{x} = \tilde{b} \tag{9.1}$$

where

$$\begin{aligned} \tilde{A} &= S_1 P_1^{-1} A P_2^{-1} S_2^{-1}, \\ \tilde{b} &= S_1 P_1^{-1} b, \\ \tilde{x} &= S_2 P_2 x, \end{aligned} \tag{9.2}$$

and where

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

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

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

We note that not all of the iterative linear solvers implemented in SUNDIALS support the full range of the above options. Similarly, some of the SUNDIALS integrators only utilize a subset of these options. Exceptions to the operators shown above are described in the documentation for each SUNLINSOL implementation, or for each SUNDIALS solver "Spils" interface.

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

```
typedef struct _generic_SUNLinearSolver *SUNLinearSolver;

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

The `_generic_SUNLinearSolver_Ops` structure is essentially a list of pointers to the various actual linear solver operations, and 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 acting on `SUNLinearSolver` objects. These routines are in fact only wrappers for 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));
}
```

Table 9.2 contains a complete list of all linear solver operations defined by the generic SUNLINSOL module. In order to support both direct and iterative linear solver types, the generic SUNLINSOL module defines linear solver routines (or arguments) that may be specific to individual use cases. As such, for each routine we specify its intended use. If a custom SUNLINSOL module is provided, the function pointers for non-required routines may be set to `NULL` to indicate that they are not provided.

A particular implementation of the SUNLINSOL module must:

Table 9.1: Identifiers associated with linear solver kernels supplied with SUNDIALS.

Linear Solver ID	Solver type	ID Value
SUNLINEARSOLVER_DIRECT	Direct solvers	0
SUNLINEARSOLVER_ITERATIVE	Iterative solvers	1
SUNLINEARSOLVER_CUSTOM	Custom solvers	2

- 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.
- Optionally, define and implement additional user-callable routines acting on the newly defined `SUNLinearSolver` (e.g., routines to set various configuration options for tuning the linear solver to a particular problem).
- Optionally, provide functions as needed for that particular implementation to access different parts in the *content* field of the newly defined `SUNLinearSolver` object (e.g., routines to return various statistics from the solver).

Each SUNLINSOL implementation included in SUNDIALS has a "type" identifier specified in enumeration and shown in Table 9.1. It is recommended that a user-supplied SUNLINSOL implementation set this identifier based on the SUNDIALS solver interface they intend to use: "Dls" interfaces require the `SUNLINEARSOLVER_DIRECT` SUNLINSOL objects and "Spils" interfaces require the `SUNLINEARSOLVER_ITERATIVE` objects.

Table 9.2: Description of the `SUNLinearSolver` operations

Name	Usage and Description
`SUNLinSolGetType`	`type = SUNLinSolGetType(LS);` Returns the type identifier for the linear solver `LS`. It is used to determine the solver type (direct, iterative, or custom) from the abstract `SUNLinearSolver` interface. This is used to assess compatibility with SUNDIALS-provided linear solver interfaces. Returned values are given in the Table 9.1.

continued on next page

Name	Usage and Description
SUNLinSolInitialize	`ier = SUNLinSolInitialize(LS);` Performs linear solver initialization (assumes that all solver-specific options have been set). This should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 9.4.
SUNLinSolSetup	`ier = SUNLinSolSetup(LS, A);` Performs any linear solver setup needed, based on an updated system SUNMATRIX `A`. This may be called frequently (e.g. with a full Newton method) or infrequently (for a modified Newton method), based on the type of integrator and/or nonlinear solver requesting the solves. This should return zero for a successful call, a positive value for a recoverable failure and a negative value for an unrecoverable failure, ideally returning one of the generic error codes listed in Table 9.4.
SUNLinSolSolve	`ier = SUNLinSolSolve(LS, A, x, b, tol);` Solves a linear system $Ax = b$. This should return zero for a successful call, a positive value for a recoverable failure and a negative value for an unrecoverable failure, ideally returning one of the generic error codes listed in Table 9.4. **Direct solvers:** can ignore the `realtype` argument `tol`. **Iterative solvers:** can ignore the SUNMATRIX input `A` since a `NULL` argument will be passed (these should instead rely on the matrix-vector product function supplied through the routine `SUNLinSolSetATimes`). These should attempt to solve to the specified `realtype` tolerance `tol` in a weighted 2-norm. If the solver does not support scaling then it should just use a 2-norm. **Custom solvers:** all arguments will be supplied, and if the solver is approximate then it should attempt to solve to the specified `realtype` tolerance `tol` in a weighted 2-norm. If the solver does not support scaling then it should just use a 2-norm.
SUNLinSolFree	`ier = SUNLinSolFree(LS);` Frees memory allocated by the linear solver. This should return zero for a successful call, and a negative value for a failure.
SUNLinSolSetATimes	`ier = SUNLinSolSetATimes(LS, A_data, ATimes);` (Iterative/Custom linear solvers only) Provides `ATimesFn` function pointer, as well as a `void *` pointer to a data structure used by this routine, to a linear solver object. SUNDIALS solvers 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. This routine should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 9.4.

continued on next page

Name	Usage and Description
SUNLinSolSetPreconditioner	`ier = SUNLinSolSetPreconditioner(LS, Pdata, Pset, Psol);` (Optional; Iterative/Custom linear solvers only) Provides `PSetupFn` and `PSolveFn` function pointers that implement the preconditioner solves P_1^{-1} and P_2^{-1} from equations (9.1)-(9.2). This routine will be called by a SUNDIALS solver, which will provide translation between the generic `Pset` and `Psol` calls and the integrator-specific and integrator- or user-supplied routines. This routine should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 9.4.
SUNLinSolSetScalingVectors	`ier = SUNLinSolSetScalingVectors(LS, s1, s2);` (Optional; Iterative/Custom linear solvers only) Sets pointers to left/right scaling vectors for the linear system solve. Here, `s1` is an NVECTOR of positive scale factors containing the diagonal of the matrix S_1 from equations (9.1)-(9.2). Similarly, `s2` is an NVECTOR containing the diagonal of S_2 from equations (9.1)-(9.2). Neither of these vectors are tested for positivity, and a `NULL` argument for either indicates that the corresponding scaling matrix is the identity. This routine should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in Table 9.4.
SUNLinSolNumIters	`its = SUNLinSolNumIters(LS);` (Optional; Iterative/Custom linear solvers only) Should return the `int` number of linear iterations performed in the last 'solve' call.
SUNLinSolResNorm	`rnorm = SUNLinSolResNorm(LS);` (Optional; Iterative/Custom linear solvers only) Should return the `realtype` final residual norm from the last 'solve' call.
SUNLinSolResid	`rvec = SUNLinSolResid(LS);` (Optional; Iterative/Custom linear solvers only) 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 function may be called by the SUNDIALS solver. This routine should return the NVECTOR containing the preconditioned initial residual vector.

continued on next page

Name	Usage and Description
SUNLinLastFlag	lflag = SUNLinLastFlag(LS); (Optional) Should return the last error flag encountered within the linear solver. This is not called by the SUNDIALS solvers directly; it allows the user to investigate linear solver issues after a failed solve.
SUNLinSolSpace	ier = SUNLinSolSpace(LS, &lrw, &liw); (Optional) Returns the storage requirements for the linear solver LS. 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. This function is advisory only, for use in determining a user's total space requirements.

9.1 Description of the client-supplied SUNLinearSolver routines

The SUNDIALS packages provide the ATimes, Pset and Psol routines utilized by the SUNLINSOL modules. These function types are defined in the header file sundials/sundials_iterative.h, and are described here in case a user wishes to interact directly with an iterative SUNLINSOL object.

ATimesFn

Definition `typedef int (*ATimesFn)(void *A_data, N_Vector v, N_Vector z);`

Purpose These functions compute the action of a matrix on a vector, performing the operation $z = Av$. Memory for z should already be allocted prior to calling this function. The vector v should be left unchanged.

Arguments A_data is a pointer to client data, the same as that supplied to SUNLinSolSetATimes.

v is the input vector to multiply.

z is the output vector computed.

Return value This routine should return 0 if successful and a non-zero value if unsuccessful.

Notes

PSetupFn

Definition `typedef int (*PSetupFn)(void *P_data)`

Purpose These functions set up any requisite problem data in preparation for calls to the corresponding PSolveFn.

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

Return value This routine should return 0 if successful and a non-zero value if unsuccessful.

Notes

`PSolveFn`

Definition
```
typedef int (*PSolveFn)(void *P_data, N_Vector r, N_Vector z,
                        realtype tol, int lr)
```

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

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

where the 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

9.2 Compatibility of SUNLinearSolver modules

We note that not all SUNLINSOL types are compatible with all SUNMATRIX and NVECTOR types provided with SUNDIALS. In Table 9.3 we show the direct linear solvers available as SUNLINSOL modules, and the compatible matrix implementations. Recall that Table 4.1 shows the compatibility between all SUNLINSOL modules and vector implementations.

Table 9.3: SUNDIALS direct 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			✓	✓

continued on next page

Linear Solver Interface	Dense Matrix	Banded Matrix	Sparse Matrix	User Supplied
User supplied	✓	✓	✓	✓

The functions within the SUNDIALS-provided `SUNLinearSolver` implementations return a common set of error codes, shown below in the Table 9.4.

Table 9.4: 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_SPFGMR)
SUNLS_QRSOL_FAIL	-9	a singular R matrix was encountered in a QR factorization (SUNLINSOL_SPGMR/SUNLINSOL_SPFGMR)
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 `ATimes` routine
SUNLS_PSET_FAIL_REC	4	a recoverable failure occurred in the `Pset` routine
SUNLS_PSOLVE_FAIL_REC	5	a recoverable failure occurred in the `Psolve` 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_SPFGMR)
SUNLS_LUFACT_FAIL	8	a singular matrix was encountered during a LU factorization (SUNLINSOL_DENSE/SUNLINSOL_BAND)

9.3 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). 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 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 SUNLINSOL_DENSE module defines dense implementations of all "direct" linear solver operations listed in Table 9.2:

- `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`

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

- `SUNDenseLinearSolver`

  This function creates and allocates memory for a dense `SUNLinearSolver`. Its arguments are an NVECTOR and SUNMATRIX, that it uses to determine the linear system size and to assess compatibility with the linear solver implementation.

  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.

  If either `A` or `y` are incompatible then this routine will return `NULL`.

  ```
  SUNLinearSolver SUNDenseLinearSolver(N_Vector y, SUNMatrix A);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_DENSE module also includes the Fortran-callable function `FSUNDenseLinSolInit(code, ier)` to initialize this SUNLINSOL_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); `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`. 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 Fortran-callable function `FSUNMassDenseLinSolInit(ier)` initializes this SUNLINSOL_DENSE module for solving mass matrix linear systems.

9.4 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). 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 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 SUNLINSOL_BAND module defines band implementations of all "direct" linear solver operations listed in Table 9.2:

- `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

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

- SUNBandLinearSolver

  This function creates and allocates memory for a band SUNLinearSolver. Its arguments are an NVECTOR and SUNMATRIX, that it uses to determine the linear system size and to assess compatibility with the linear solver implementation.

  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.

  If either A or y are incompatible then this routine will return NULL.

  ```
  SUNLinearSolver SUNBandLinearSolver(N_Vector y, SUNMatrix A);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_BAND module also includes the Fortran-callable function FSUNBandLinSolInit(code, ier) to initialize this SUNLINSOL_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); 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. 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 Fortran-callable function FSUNMassBandLinSolInit(ier) initializes this SUNLINSOL_BAND module for solving mass matrix linear systems.

9.5 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). 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 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 SUNLINSOL_LAPACKDENSE module defines dense implementations of all "direct" linear solver operations listed in Table 9.2:

- 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

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

- SUNLapackDense

  This function creates and allocates memory for a LAPACK dense SUNLinearSolver. Its arguments are an NVECTOR and SUNMATRIX, that it uses to determine the linear system size and to assess compatibility with the linear solver implementation.

  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.

  If either A or y are incompatible then this routine will return NULL.

  ```
  SUNLinearSolver SUNLapackDense(N_Vector y, SUNMatrix A);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_LAPACKDENSE module also includes the Fortran-callable function `FSUNLapackDenseInit(code, ier)` to initialize this SUNLINSOL_LAPACKDENSE 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); `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`. 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 Fortran-callable function `FSUNMassLapackDenseInit(ier)` initializes this SUNLINSOL_LAPACKDENSE module for solving mass matrix linear systems.

9.6 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). 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 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 SUNLINSOL_LAPACKBAND module defines band implementations of all "direct" linear solver operations listed in Table 9.2:

- `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`

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

- `SUNLapackBand`

  This function creates and allocates memory for a LAPACK band `SUNLinearSolver`. Its arguments are an NVECTOR and SUNMATRIX, that it uses to determine the linear system size and to assess compatibility with the linear solver implementation.

  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.

  If either `A` or `y` are incompatible then this routine will return `NULL`.

  ```
  SUNLinearSolver SUNLapackBand(N_Vector y, SUNMatrix A);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_LAPACKBAND module also includes the Fortran-callable function `FSUNLapackBandInit(code, ier)` to initialize this SUNLINSOL_LAPACKBAND 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); `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`. 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 Fortran-callable function `FSUNMassLapackBandInit(ier)` initializes this SUNLINSOL_LAPACKBAND module for solving mass matrix linear systems.

9.7 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). 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, 11]. 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 header file to include when using this module is `sunlinsol/sunlinsol_klu.h`. The installed module library to link to is `libsundials_sunlinsolklu.`*`lib`* where *`.lib`* is typically `.so` for shared libraries and `.a` for static libraries.
The SUNLINSOL_KLU module defines implementations of all "direct" linear solver operations listed in Table 9.2:

- `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`

The module SUNLINSOL_KLU provides the following additional user-callable routines:

- `SUNKLU`

  This constructor function creates and allocates memory for a SUNLINSOL_KLU object. Its arguments are an NVECTOR and SUNMATRIX, that it uses to determine the linear system size and to assess compatibility with the linear solver implementation.

  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.

  If either `A` or `y` are incompatible then this routine will return `NULL`.

  ```
  SUNLinearSolver SUNKLU(N_Vector y, SUNMatrix A);
  ```

- `SUNKLUReInit`

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

  The `reinit_type` argument governs the level of reinitialization. The allowed values are:

  - `SUNKLU_REINIT_FULL` – The Jacobian matrix will be destroyed and a new one will be allocated based on the `nnz` value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup.
  - `SUNKLU_REINIT_PARTIAL` – Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of `nnz` given in the sparse matrix provided to the original constructor routine (or the previous `SUNKLUReInit` call).

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

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.

```
int SUNKLUReInit(SUNLinearSolver S, SUNMatrix A,
                 sunindextype nnz, int reinit_type);
```

- SUNKLUSetOrdering

  This function sets the ordering used by KLU for reducing fill in the linear solve. Options for ordering_choice are:

  0 AMD,

  1 COLAMD, and

  2 the natural ordering.

  The default is 1 for COLAMD.

  The return values from this function are SUNLS_MEM_NULL (S is NULL), SUNLS_ILL_INPUT (invalid ordering_choice), or SUNLS_SUCCESS.

  ```
  int SUNKLUSetOrdering(SUNLinearSolver S, int ordering_choice);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_KLU module also includes the Fortran-callable function FSUNKLUInit(code, ier) to initialize this SUNLINSOL_KLU 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); 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. 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 Fortran-callable function FSUNMassKLUInit(ier) initializes this SUNLINSOL_KLU module for solving mass matrix linear systems.

The SUNKLUReInit and SUNKLUSetOrdering routines also support Fortran interfaces for the system and mass matrix solvers:

- FSUNKLUReInit(code, NNZ, reinit_type, ier) – NNZ should be commensurate with a C long int and reinit_type should be commensurate with a C int (1 = 'FULL', 2 = 'PARTIAL')
- FSUNMassKLUReInit(NNZ, reinit_type, ier)
- FSUNKLUSetOrdering(code, ordering, ier) – ordering should be commensurate with a C int
- FSUNMassKLUSetOrdering(ordering, ier)

9.8 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. 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;
  superlumt_options_t *options;
};
```

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

last_flag - last error return flag from internal function evaluations,

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

A, AC, L, U, B - `SuperMatrix` pointers used in solve,

Gstat - `GStat_t` object used in solve,

perm_r, perm_c - permutation arrays used in solve,

N - size of the linear system,

num_threads - number of OpenMP/Pthreads threads to use,

diag_pivot_thresh - threshold on diagonal pivoting,

ordering - flag for which reordering algorithm to use,

options - pointer to 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, 27, 12]. 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 header file to include when using this module is `sunlinsol/sunlinsol_superlumt.h`. The installed module library to link to is `libsundials_sunlinsolsuperlumt.`*`lib`* where *`.lib`* is typically `.so` for shared libraries and `.a` for static libraries.
The SUNLINSOL_SUPERLUMT module defines implementations of all "direct" linear solver operations listed in Table 9.2:

- `SUNLinSolGetType_SuperLUMT`
- `SUNLinSolInitialize_SuperLUMT` – this sets the `first_factorize` flag to 1 and resets the internal SUPERLUMT statistics variables.
- `SUNLinSolSetup_SuperLUMT` – this performs either a *LU* factorization or refactorization of the input matrix.
- `SUNLinSolSolve_SuperLUMT` – this calls the appropriate SUPERLUMT solve routine to utilize the *LU* factors to solve the linear system.
- `SUNLinSolLastFlag_SuperLUMT`
- `SUNLinSolSpace_SuperLUMT` – this only returns information for the storage within the solver *interface*, i.e. storage for the integers `last_flag` and `first_factorize`. For additional space requirements, see the SUPERLUMT documentation.
- `SUNLinSolFree_SuperLUMT`

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

- `SUNSuperLUMT`

  This constructor function creates and allocates memory for a SUNLINSOL_SUPERLUMT object. Its arguments are an NVECTOR, a SUNMATRIX, and a desired number of threads (OpenMP or Pthreads, depending on how SUPERLUMT was installed) to use during the factorization steps. This routine analyzes the input matrix and vector to determine the linear system size and to assess compatibility with the SUPERLUMT library.

  This routine will perform consistency checks to ensure that it is called with consistent NVECTOR and SUNMATRIX implementations. These are currently limited to the SUNMATRIX_SPARSE matrix type (using either CSR or CSC storage formats) and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

  If either `A` or `y` are incompatible then this routine will return `NULL`. The `num_threads` argument is not checked and is passed directly to SUPERLUMT routines.

  ```
  SUNLinearSolver SUNSuperLUMT(N_Vector y, SUNMatrix A, int num_threads);
  ```

- `SUNSuperLUMTSetOrdering`

  This function sets the ordering used by SUPERLUMT for reducing fill in the linear solve. Options for `ordering_choice` 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.

The return values from this function are `SUNLS_MEM_NULL` (`S` is `NULL`), `SUNLS_ILL_INPUT` (invalid `ordering_choice`), or `SUNLS_SUCCESS`.

```
int SUNSuperLUMTSetOrdering(SUNLinearSolver S, int ordering_choice);
```

For solvers that include a Fortran interface module, the SUNLINSOL_SUPERLUMT module also includes the Fortran-callable function `FSUNSuperLUMTInit(code, num_threads, ier)` to initialize this SUNLINSOL_SUPERLUMT 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); `num_threads` is the desired number of OpenMP/Pthreads threads to use in the factorization; `ier` is an error return flag equal to 0 for success and -1 for failure. All of these arguments should be declared so as to match C type `int`. 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 Fortran-callable function `FSUNMassSuperLUMTInit(num_threads, ier)` initializes this SUNLINSOL_SUPERLUMT module for solving mass matrix linear systems.

The `SUNSuperLUMTSetOrdering` routine also supports Fortran interfaces for the system and mass matrix solvers:

- `FSUNSuperLUMTSetOrdering(code, ordering, ier)` – `ordering` should be commensurate with a C `int`
- `FSUNMassSuperLUMTSetOrdering(ordering, ier)`

9.9 The SUNLinearSolver_SPGMR implementation

The SPGMR (Scaled, Preconditioned, Generalized Minimum Residual [33]) 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`).

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 $v_1, \ldots, v_{\texttt{maxl}+1}$, stored in `V[0]`, ..., `V[maxl]`. Each v_i is a vector of type NVECTOR.,

Hes - the $(\texttt{maxl}+1) \times \texttt{maxl}$ Hessenberg matrix. It is stored row-wise so that the (i,j)th element is given by `Hes[i][j]`.,

givens - a length `2*maxl` array which represents the Givens rotation matrices that arise in the GMRES algorithm. These matrices are $F_0, F_1, \ldots, F_j$, where $F_i = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & c_i & -s_i & & \\ & & & s_i & c_i & & \\ & & & & & 1 & \\ & & & & & & \ddots \\ & & & & & & & 1 \end{bmatrix}$, are represented in the `givens` vector as `givens[0]` $= c_0$, `givens[1]` $= s_0$, `givens[2]` $= c_1$, `givens[3]` $= s_1$, ... `givens[2j]` $= c_j$, `givens[2j+1]` $= s_j$.,

xcor - a vector which holds the scaled, preconditioned correction to the initial guess,

yg - a length `(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 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 SUNLINSOL_SPGMR module defines implementations of all "iterative" linear solver operations listed in Table 9.2:

- `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`

The module SUNLINSOL_SPGMR provides the following additional user-callable routines:

- `SUNSPGMR`

  This constructor function creates and allocates memory for a SPGMR `SUNLinearSolver`. Its arguments are an NVECTOR, the desired type of preconditioning, and the number of Krylov basis vectors to use.

  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`.

  A `maxl` argument that is ≤ 0 will result in the default value (5).

  Allowable inputs for `pretype` are `PREC_NONE` (0), `PREC_LEFT` (1), `PREC_RIGHT` (2) and `PREC_BOTH` (3); any other integer input will result in the default (no preconditioning). We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) 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.

  ```
  SUNLinearSolver SUNSPGMR(N_Vector y, int pretype, int maxl);
  ```

- SUNSPGMRSetPrecType

  This function updates the type of preconditioning to use. Supported values are PREC_NONE (0), PREC_LEFT (1), PREC_RIGHT (2) and PREC_BOTH (3).

  This routine will return with one of the error codes SUNLS_ILL_INPUT (illegal pretype), SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

  ```
  int SUNSPGMRSetPrecType(SUNLinearSolver S, int pretype);
  ```

- SUNSPGMRSetGSType

  This function sets the type of Gram-Schmidt orthogonalization to use. Supported values are MODIFIED_GS (1) and CLASSICAL_GS (2). Any other integer input will result in a failure, returning error code SUNLS_ILL_INPUT.

  This routine will return with one of the error codes SUNLS_ILL_INPUT (illegal gstype), SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

  ```
  int SUNSPGMRSetGSType(SUNLinearSolver S, int gstype);
  ```

- SUNSPGMRSetMaxRestarts

  This function sets the number of GMRES restarts to allow. A negative input will result in the default of 0.

  This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

  ```
  int SUNSPGMRSetMaxRestarts(SUNLinearSolver S, int maxrs);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_SPGMR module also includes the Fortran-callable function FSUNSPGMRInit(code, pretype, maxl, ier) to initialize this SUNLINSOL_SPGMR 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); pretype and maxl are the same as for the C function SUNSPGMR; ier is an error return flag equal to 0 for success and -1 for failure. All of these input arguments should be declared so as to match C type int. This routine must be called *after* the NVECTOR object has been initialized. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function FSUNMassSPGMRInit(pretype, maxl, ier) initializes this SUNLINSOL_SPGMR module for solving mass matrix linear systems.

The SUNSPGMRSetPrecType, SUNSPGMRSetGSType and SUNSPGMRSetMaxRestarts routines also support Fortran interfaces for the system and mass matrix solvers (all arguments should be commensurate with a C int):

- FSUNSPGMRSetGSType(code, gstype, ier)
- FSUNMassSPGMRSetGSType(gstype, ier)
- FSUNSPGMRSetPrecType(code, pretype, ier)
- FSUNMassSPGMRSetPrecType(pretype, ier)
- FSUNSPGMRSetMaxRS(code, maxrs, ier)
- FSUNMassSPGMRSetMaxRS(maxrs, ier)

9.10 The SUNLinearSolver_SPFGMR implementation

The SPFGMR (Scaled, Preconditioned, Flexible, Generalized Minimum Residual [32]) implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_SPFGMR, 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 other Krylov iterative linear

solvers supplied with SUNDIALS, FGMRES is specifically designed to work with a changing preconditioner (e.g. from an iterative method).

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 $v_1, \ldots, v_{\texttt{maxl}+1}$, stored in `V[0]`, ..., `V[maxl]`. Each v_i is a vector of type NVECTOR.,

Z - the array of preconditioned Krylov basis vectors $z_1, \ldots, z_{\texttt{maxl}+1}$, stored in `Z[0]`, ..., `Z[maxl]`. Each z_i is a vector of type NVECTOR.,

Hes - the $(\texttt{maxl}+1) \times \texttt{maxl}$ Hessenberg matrix. It is stored row-wise so that the (i,j)th element is given by `Hes[i][j]`.,

givens - a length `2*maxl` array which represents the Givens rotation matrices that arise in the FGMRES algorithm. These matrices are $F_0, F_1, \ldots, F_j$, where $F_i = \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_i & -s_i & & & \\ & & & s_i & c_i & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix}$, are represented in the `givens` vector as `givens[0]` $= c_0$, `givens[1]` $= s_0$, `givens[2]` $= c_1$, `givens[3]` $= s_1$, ... `givens[2j]` $= c_j$, `givens[2j+1]` $= s_j$.,

xcor - a vector which holds the scaled, preconditioned correction to the initial guess,

yg - a length `(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 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 SUNLINSOL_SPFGMR module defines implementations of all "iterative" linear solver operations listed in Table 9.2:

- `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

The module SUNLINSOL_SPFGMR provides the following additional user-callable routines:

- SUNSPFGMR

  This constructor function creates and allocates memory for a SPFGMR SUNLinearSolver. Its arguments are an NVECTOR, a flag indicating to use preconditioning, and the number of Krylov basis vectors to use.

  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.

  A maxl argument that is ≤ 0 will result in the default value (5).

  Since the FGMRES algorithm is designed to only support right preconditioning, then any of the pretype inputs PREC_LEFT (1), PREC_RIGHT (2), or PREC_BOTH (3) will result in use of PREC_RIGHT; any other integer input will result in the default (no preconditioning). We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS). While it is possible to use a right-preconditioned SUNLINSOL_SPFGMR object for these packages, this use mode is not supported and may result in inferior performance.

  ```
  SUNLinearSolver SUNSPFGMR(N_Vector y, int pretype, int maxl);
  ```

- SUNSPFGMRSetPrecType

  This function updates the flag indicating use of preconditioning. Since the FGMRES algorithm is designed to only support right preconditioning, then any of the pretype inputs PREC_LEFT (1), PREC_RIGHT (2), or PREC_BOTH (3) will result in use of PREC_RIGHT; any other integer input will result in the default (no preconditioning).

  This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

  ```
  int SUNSPFGMRSetPrecType(SUNLinearSolver S, int pretype);
  ```

- SUNSPFGMRSetGSType

  This function sets the type of Gram-Schmidt orthogonalization to use. Supported values are MODIFIED_GS (1) and CLASSICAL_GS (2). Any other integer input will result in a failure, returning error code SUNLS_ILL_INPUT.

  This routine will return with one of the error codes SUNLS_ILL_INPUT (illegal gstype), SUNLS_MEM_NULL (S is NULL), or SUNLS_SUCCESS.

  ```
  int SUNSPFGMRSetGSType(SUNLinearSolver S, int gstype);
  ```

- SUNSPFGMRSetMaxRestarts

  This function sets the number of FGMRES restarts to allow. A negative input will result in the default of 0.

  This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

  ```
  int SUNSPFGMRSetMaxRestarts(SUNLinearSolver S, int maxrs);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_SPFGMR module also includes the Fortran-callable function `FSUNSPFGMRInit(code, pretype, maxl, ier)` to initialize this SUNLINSOL_SPFGMR 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); `pretype` and `maxl` are the same as for the C function `SUNSPFGMR`; `ier` is an error return flag equal to 0 for success and -1 for failure. All of these input arguments should be declared so as to match C type `int`. This routine must be called *after* the NVECTOR object has been initialized. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function `FSUNMassSPFGMRInit(pretype, maxl, ier)` initializes this SUNLINSOL_SPFGMR module for solving mass matrix linear systems.

The `SUNSPFGMRSetPrecType`, `SUNSPFGMRSetGSType`, and `SUNSPFGMRSetMaxRestarts` routines also support Fortran interfaces for the system and mass matrix solvers (all arguments should be commensurate with a C `int`):

- `FSUNSPFGMRSetGSType(code, gstype, ier)`
- `FSUNMassSPFGMRSetGSType(gstype, ier)`
- `FSUNSPFGMRSetPrecType(code, pretype, ier)`
- `FSUNMassSPFGMRSetPrecType(pretype, ier)`
- `FSUNSPFGMRSetMaxRS(code, maxrs, ier)`
- `FSUNMassSPFGMRSetMaxRS(maxrs, ier)`

9.11 The SUNLinearSolver_SPBCGS implementation

The SPBCGS (Scaled, Preconditioned, Bi-Conjugate Gradient, Stabilized [36]) 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.

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 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 SUNLINSOL_SPBCGS module defines implementations of all "iterative" linear solver operations listed in Table 9.2:

- `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

The module SUNLINSOL_SPBCGS provides the following additional user-callable routines:

- SUNSPBCGS

  This constructor function creates and allocates memory for a SPBCGS SUNLinearSolver. Its arguments are an NVECTOR, the desired type of preconditioning, and the number of linear iterations to allow.

  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.

  A maxl argument that is ≤ 0 will result in the default value (5).

  Allowable inputs for pretype are PREC_NONE (0), PREC_LEFT (1), PREC_RIGHT (2) and PREC_BOTH (3); any other integer input will result in the default (no preconditioning). We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) 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.

  ```
  SUNLinearSolver SUNSPBCGS(N_Vector y, int pretype, int maxl);
  ```

- SUNSPBCGSSetPrecType

  This function updates the type of preconditioning to use. Supported values are PREC_NONE (0), PREC_LEFT (1), PREC_RIGHT (2), and PREC_BOTH (3).

  This routine will return with one of the error codes SUNLS_ILL_INPUT (illegal pretype), SUNLS_MEM_NULL (S is NULL), or SUNLS_SUCCESS.

  ```
  int SUNSPBCGSSetPrecType(SUNLinearSolver S, int pretype);
  ```

- SUNSPBCGSSetMaxl

  This function updates the number of linear solver iterations to allow.

  A maxl argument that is ≤ 0 will result in the default value (5).

  This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

  ```
  int SUNSPBCGSSetMaxl(SUNLinearSolver S, int maxl);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_SPBCGS module also includes the Fortran-callable function FSUNSPBCGSInit(code, pretype, maxl, ier) to initialize this SUNLINSOL_SPBCGS 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); pretype and maxl are the same as for the C function SUNSPBCGS; ier is an error return flag equal to 0 for success and -1 for failure. All of these input arguments should be declared so as to match C type int. This routine must be called *after* the NVECTOR object has been initialized. Additionally, when using ARKODE with a non-identity

mass matrix, the Fortran-callable function `FSUNMassSPBCGSInit(pretype, maxl, ier)` initializes this SUNLINSOL_SPBCGS module for solving mass matrix linear systems.

The `SUNSPBCGSSetPrecType` and `SUNSPBCGSSetMaxl` routines also support Fortran interfaces for the system and mass matrix solvers (all arguments should be commensurate with a C `int`):

- `FSUNSPBCGSSetPrecType(code, pretype, ier)`
- `FSUNMassSPBCGSSetPrecType(pretype, ier)`
- `FSUNSPBCGSSetMaxl(code, maxl, ier)`
- `FSUNMassSPBCGSSetMaxl(maxl, ier)`

9.12 The SUNLinearSolver_SPTFQMR implementation

The SPTFQMR (Scaled, Preconditioned, Transpose-Free Quasi-Minimum Residual [14]) 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.

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 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 SUNLINSOL_SPTFQMR module defines implementations of all "iterative" linear solver operations listed in Table 9.2:

- `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

The module SUNLINSOL_SPTFQMR provides the following additional user-callable routines:

- SUNSPTFQMR

  This constructor function creates and allocates memory for a SPTFQMR SUNLinearSolver. Its arguments are an NVECTOR, the desired type of preconditioning, and the number of linear iterations to allow.

  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.

  A maxl argument that is ≤ 0 will result in the default value (5).

  Allowable inputs for pretype are PREC_NONE (0), PREC_LEFT (1), PREC_RIGHT (2) and PREC_BOTH (3); any other integer input will result in the default (no preconditioning). We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL). While it is possible to configure a SUNLINSOL_SPTFQMR object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.

  ```
  SUNLinearSolver SUNSPTFQMR(N_Vector y, int pretype, int maxl);
  ```

- SUNSPTFQMRSetPrecType

  This function updates the type of preconditioning to use. Supported values are PREC_NONE (0), PREC_LEFT (1), PREC_RIGHT (2), and PREC_BOTH (3).

  This routine will return with one of the error codes SUNLS_ILL_INPUT (illegal pretype), SUNLS_MEM_NULL (S is NULL), or SUNLS_SUCCESS.

  ```
  int SUNSPTFQMRSetPrecType(SUNLinearSolver S, int pretype);
  ```

- SUNSPTFQMRSetMaxl

  This function updates the number of linear solver iterations to allow.

  A maxl argument that is ≤ 0 will result in the default value (5).

  This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

  ```
  int SUNSPTFQMRSetMaxl(SUNLinearSolver S, int maxl);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_SPTFQMR module also includes the Fortran-callable function FSUNSPTFQMRInit(code, pretype, maxl, ier) to initialize this SUNLINSOL_SPTFQMR 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); pretype and maxl are the same as for the C function SUNSPTFQMR; ier is an error return flag equal to 0 for success and -1 for failure. All of these input arguments should be declared so as to match C type int. This routine must be called *after* the NVECTOR object has been initialized. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function FSUNMassSPTFQMRInit(pretype, maxl, ier) initializes this SUNLINSOL_SPTFQMR module for solving mass matrix linear systems.

The SUNSPTFQMRSetPrecType and SUNSPTFQMRSetMaxl routines also support Fortran interfaces for the system and mass matrix solvers (all arguments should be commensurate with a C int):

- FSUNSPTFQMRSetPrecType(code, pretype, ier)

- `FSUNMassSPTFQMRSetPrecType(pretype, ier)`
- `FSUNSPTFQMRSetMaxl(code, maxl, ier)`
- `FSUNMassSPTFQMRSetMaxl(maxl, ier)`

9.13 The SUNLinearSolver_PCG implementation

The PCG (Preconditioned Conjugate Gradient [15]) 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

$$\tilde{A}\tilde{x} = \tilde{b} \tag{9.3}$$

where

$$\begin{aligned} \tilde{A} &= SP^{-1}AP^{-1}S, \\ \tilde{b} &= SP^{-1}b, \\ \tilde{x} &= S^{-1}Px. \end{aligned} \tag{9.4}$$

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.

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 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 SUNLINSOL_PCG module defines implementations of all "iterative" linear solver operations listed in Table 9.2:

- `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`

The module SUNLINSOL_PCG provides the following additional user-callable routines:

- `SUNPCG`

  This constructor function creates and allocates memory for a PCG `SUNLinearSolver`. Its arguments are an NVECTOR, a flag indicating to use preconditioning, and the number of linear iterations to allow.

  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`.

  A `maxl` argument that is ≤ 0 will result in the default value (5).

  Since the PCG algorithm is designed to only support symmetric preconditioning, then any of the `pretype` inputs `PREC_LEFT` (1), `PREC_RIGHT` (2), or `PREC_BOTH` (3) will result in use of the symmetric preconditioner; any other integer input will result in the default (no preconditioning). Although some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL), PCG should *only* be used with these packages when the linear systems are known to be *symmetric*. Since the scaling of matrix rows and columns must be identical in a symmetric matrix, symmetric preconditioning should work appropriately even for packages designed with one-sided preconditioning in mind.

  ```
  SUNLinearSolver SUNPCG(N_Vector y, int pretype, int maxl);
  ```

- `SUNPCGSetPrecType`

  This function updates the flag indicating use of preconditioning. As above, any one of the input values, `PREC_LEFT` (1), `PREC_RIGHT` (2), or `PREC_BOTH` (3) will enable preconditioning; `PREC_NONE` (0) disables preconditioning.

  This routine will return with one of the error codes `SUNLS_ILL_INPUT` (illegal `pretype`), `SUNLS_MEM_NULL` (`S` is `NULL`), or `SUNLS_SUCCESS`.

  ```
  int SUNPCGSetPrecType(SUNLinearSolver S, int pretype);
  ```

- `SUNPCGSetMaxl`

  This function updates the number of linear solver iterations to allow.

  A `maxl` argument that is ≤ 0 will result in the default value (5).

  This routine will return with one of the error codes `SUNLS_MEM_NULL` (`S` is `NULL`) or `SUNLS_SUCCESS`.

  ```
  int SUNPCGSetMaxl(SUNLinearSolver S, int maxl);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_PCG module also includes the Fortran-callable function `FSUNPCGInit(code, pretype, maxl, ier)` to initialize this SUNLINSOL_PCG 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); `pretype` and `maxl` are the same as for the C function `SUNPCG`; `ier` is an error return flag equal to 0 for success and -1 for failure. All of these input arguments should be declared so as to match C type `int`. This routine must be called *after* the NVECTOR object has been initialized. Additionally, when using ARKODE with a non-identity mass matrix, the Fortran-callable function `FSUNMassPCGInit(pretype, maxl, ier)` initializes this SUNLINSOL_PCG module for solving mass matrix linear systems.

The `SUNPCGSetPrecType` and `SUNPCGSetMaxl` routines also support Fortran interfaces for the system and mass matrix solvers (all arguments should be commensurate with a C `int`):

- `FSUNPCGSetPrecType(code, pretype, ier)`
- `FSUNMassPCGSetPrecType(pretype, ier)`
- `FSUNPCGSetMaxl(code, maxl, ier)`
- `FSUNMassPCGSetMaxl(maxl, ier)`

9.14 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*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.

9.15 SUNLinearSolver functions used by CVODES

In Table 9.5 below, we list the linear solver functions in the SUNLINSOL module used within the CVODES package. The table also shows, for each function, which of the code modules uses the function. In general, the main CVODES integrator considers three categories of linear solvers, *direct*, *iterative* and *custom*, with interfaces accessible in the CVODES header files `cvodes_direct.h` (CVDLS), `cvodes_spils.h` (CVSPILS) and `cvodes_customls.h` (CVCLS), respectively. Hence, the the table columns reference the use of SUNLINSOL functions by each of these solver interfaces.

As with the SUNMATRIX module, we emphasize that the CVODES user does not need to know detailed usage of linear solver functions by the CVODES code modules in order to use CVODES. The information is presented as an implementation detail for the interested reader.

Table 9.5: List of linear solver functions usage by CVODES code modules

	CVDLS	CVSPILS	CVCLS
SUNLinSolGetType	✓	✓	†
SUNLinSolSetATimes		✓	†
SUNLinSolSetPreconditioner		✓	†
SUNLinSolSetScalingVectors		✓	†
SUNLinSolInitialize	✓	✓	✓
SUNLinSolSetup	✓	✓	✓
SUNLinSolSolve	✓	✓	✓
SUNLinSolNumIters		✓	†
SUNLinSolResNorm		✓	†
SUNLinSolResid		✓	†
SUNLinSolLastFlag			
SUNLinSolFree	✓	✓	✓
SUNLinSolSpace	†	†	†

The linear solver functions listed in Table 9.2 with a † symbol are optionally used, in that these are only called if they are implemented in the SUNLINSOL module that is being used (i.e. their function

pointers are non-`NULL`). Also, although CVODES does not call the `SUNLinSolLastFlag` directly, this routine is available for users to query linear solver issues directly.

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 *installdir* defaults to `/usr/local` and can be changed by setting the `CMAKE_INSTALL_PREFIX` variable. Support for FORTRAN and all other options are disabled.

CMake can be used from the command line with the `cmake` command, or from a `curses`-based GUI by using the `ccmake` command. Examples for using both methods will be presented. For the examples shown it is assumed that there is a top level SUNDIALS directory with appropriate source, build and install directories:

```
% mkdir (...)sundials/instdir
% mkdir (...)sundials/builddir
% cd (...)sundials/builddir
```

Building with the GUI

Using CMake with the GUI follows this general process:

- Select and modify values, run configure (`c` key)
- New values are denoted with an asterisk
- To set a variable, move the cursor to the variable and press enter
  - If it is a boolean (ON/OFF) it will toggle the value
  - If it is string or file, it will allow editing of the string
  - For file and directories, the `<tab>` key can be used to complete

- Repeat until all values are set as desired and the generate option is available (`g` key)
- Some variables (advanced variables) are not visible right away
- To see advanced variables, toggle to advanced mode (`t` key)
- To search for a variable press `/` key, and to repeat the search, press the `n` key

To build the default configuration using the GUI, from the *builddir* enter the ccmake command and point to the *solverdir*:

```
% ccmake ../solverdir
```

The default configuration screen is shown in Figure A.1.

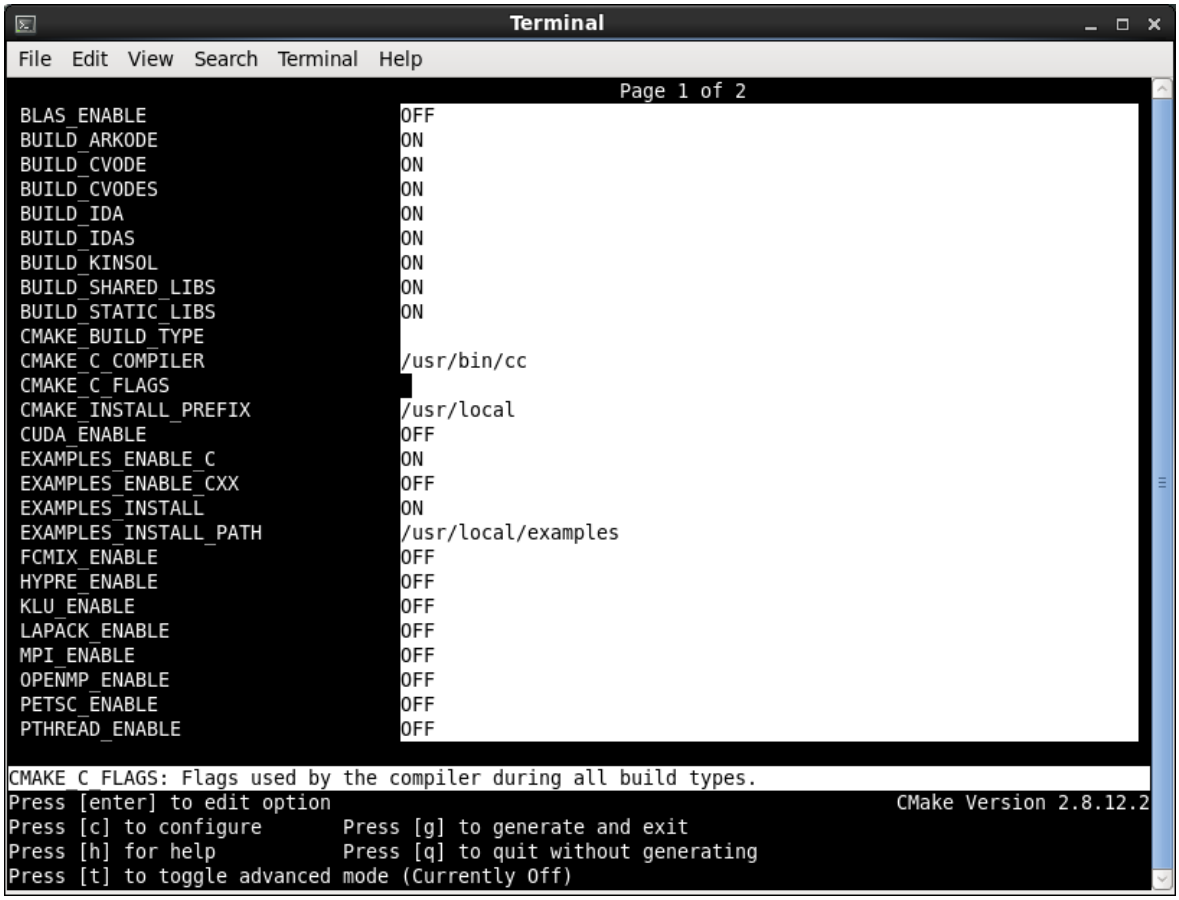


Figure A.1: Default configuration screen. Note: Initial screen is empty. To get this default configuration, press 'c' repeatedly (accepting default values denoted with asterisk) until the 'g' option is available.

The default *instdir* for both SUNDIALS and corresponding examples can be changed by setting the `CMAKE_INSTALL_PREFIX` and the `EXAMPLES_INSTALL_PATH` as shown in figure A.2.

Pressing the (`g` key) will generate makefiles including all dependencies and all rules to build SUNDIALS on this system. Back at the command prompt, you can now run:

```
% make
```

To install SUNDIALS in the installation directory specified in the configuration, simply run:

```
% make install
```

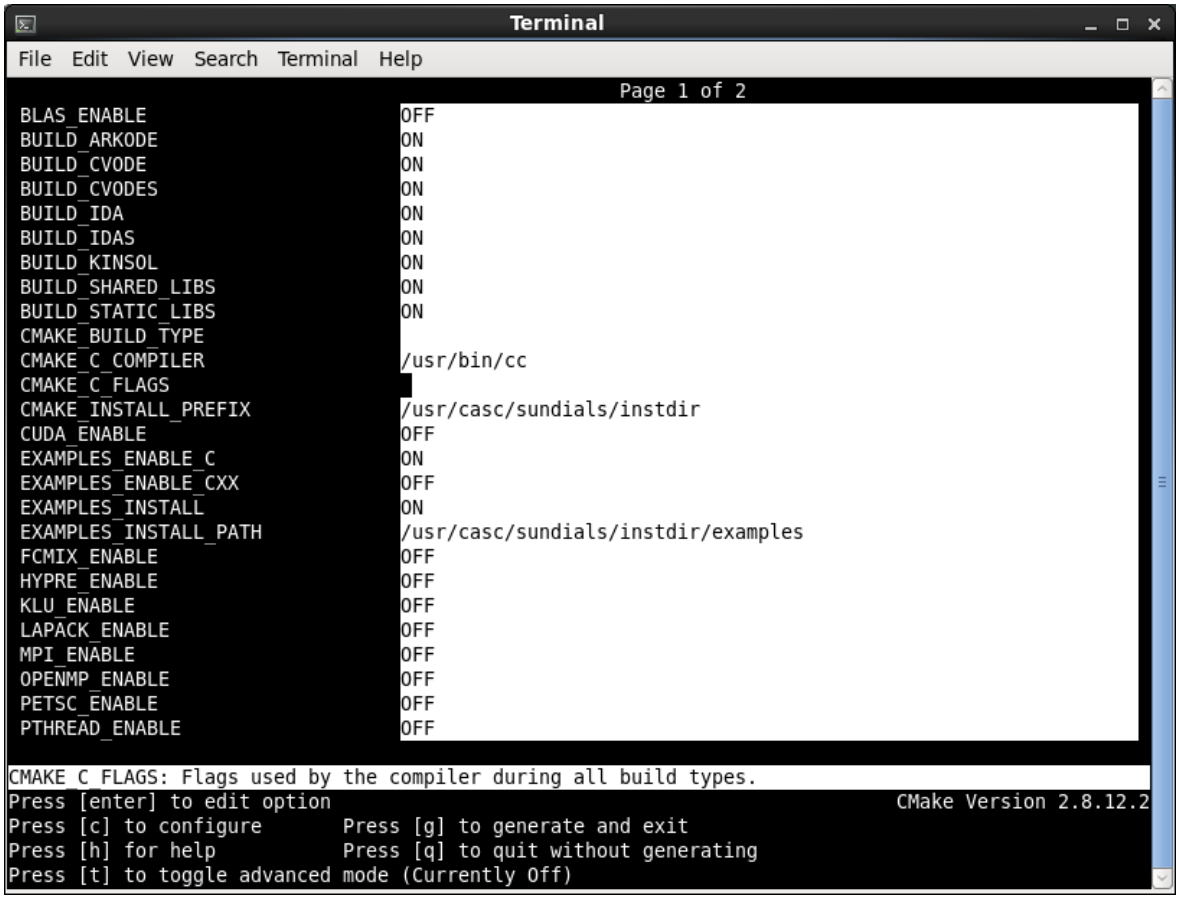


Figure A.2: Changing the *instdir* for SUNDIALS and corresponding examples

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

continued on next page

continued from last page

	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_iterative.h	sundials/sundials_direct.h
		sundials/sundials_dense.h	sundials/sundials_band.h
		sundials/sundials_matrix.h	sundials/sundials_version.h
		sundials/sundials_linearsolver.h	sundials/sundials_mpi_types.h
NVECTOR_SERIAL	Libraries	libsundials_nvecserial.*lib*	libsundials_fnvecserial.a
	Header files	nvector/nvector_serial.h	
NVECTOR_PARALLEL	Libraries	libsundials_nvecparallel.*lib*	libsundials_fnvecparallel.a
	Header files	nvector/nvector_parallel.h	
NVECTOR_OPENMP	Libraries	libsundials_nvecopenmp.*lib*	libsundials_fnvecopenmp.a
	Header files	nvector/nvector_openmp.h	
NVECTOR_PTHREADS	Libraries	libsundials_nvecpthreads.*lib*	libsundials_fnvecpthreads.a
	Header files	nvector/nvector_pthreads.h	
NVECTOR_PARHYP	Libraries	libsundials_nvecparhyp.*lib*	
	Header files	nvector/nvector_parhyp.h	
NVECTOR_PETSC	Libraries	libsundials_nvecpetsc.*lib*	
	Header files	nvector/nvector_petsc.h	
NVECTOR_CUDA	Libraries	libsundials_nveccuda.*lib*	
	Libraries	libsundials_nvecmpicuda.*lib*	
	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.*lib*	
	Libraries	libsundials_nveccudampiraja.*lib*	
	Header files	nvector/nvector_raja.h	
		nvector/nvector_mpiraja.h	
		nvector/raja/Vector.hpp	
SUNMATRIX_BAND	Libraries	libsundials_sunmatrixband.*lib*	
		libsundials_fsunmatrixband.a	
	Header files	sunmatrix/sunmatrix_band.h	
SUNMATRIX_DENSE	Libraries	libsundials_sunmatrixdense.*lib*	
		libsundials_fsunmatrixdense.a	
	Header files	sunmatrix/sunmatrix_dense.h	
SUNMATRIX_SPARSE	Libraries	libsundials_sunmatrixsparse.*lib*	
		libsundials_fsunmatrixsparse.a	
	Header files	sunmatrix/sunmatrix_sparse.h	
SUNLINSOL_BAND	Libraries	libsundials_sunlinsolband.*lib*	

continued on next page

continued from last page			
		libsundials_fsunlinsolband.a	
	Header files	sunlinsol/sunlinsol_band.h	
SUNLINSOL_DENSE	Libraries	libsundials_sunlinsoldense.*lib* libsundials_fsunlinsoldense.a	
	Header files	sunlinsol/sunlinsol_dense.h	
SUNLINSOL_KLU	Libraries	libsundials_sunlinsolklu.*lib* libsundials_fsunlinsolklu.a	
	Header files	sunlinsol/sunlinsol_klu.h	
SUNLINSOL_LAPACKBAND	Libraries	libsundials_sunlinsollapackband.*lib* libsundials_fsunlinsollapackband.a	
	Header files	sunlinsol/sunlinsol_lapackband.h	
SUNLINSOL_LAPACKDENSE	Libraries	libsundials_sunlinsollapackdense.*lib* libsundials_fsunlinsollapackdense.a	
	Header files	sunlinsol/sunlinsol_lapackdense.h	
SUNLINSOL_PCG	Libraries	libsundials_sunlinsolpcg.*lib* libsundials_fsunlinsolpcg.a	
	Header files	sunlinsol/sunlinsol_pcg.h	
SUNLINSOL_SPBCGS	Libraries	libsundials_sunlinsolspbcgs.*lib* libsundials_fsunlinsolspbcgs.a	
	Header files	sunlinsol/sunlinsol_spbcgs.h	
SUNLINSOL_SPFGMR	Libraries	libsundials_sunlinsolspfgmr.*lib* libsundials_fsunlinsolspfgmr.a	
	Header files	sunlinsol/sunlinsol_spfgmr.h	
SUNLINSOL_SPGMR	Libraries	libsundials_sunlinsolspgmr.*lib* libsundials_fsunlinsolspgmr.a	
	Header files	sunlinsol/sunlinsol_spgmr.h	
SUNLINSOL_SPTFQMR	Libraries	libsundials_sunlinsolsptfqmr.*lib* libsundials_fsunlinsolsptfqmr.a	
	Header files	sunlinsol/sunlinsol_sptfqmr.h	
SUNLINSOL_SUPERLUMT	Libraries	libsundials_sunlinsolsuperlumt.*lib* libsundials_fsunlinsolsuperlumt.a	
	Header files	sunlinsol/sunlinsol_superlumt.h	
CVODE	Libraries	libsundials_cvode.*lib*	libsundials_fcvode.a
	Header files	cvode/cvode.h cvode/cvode_direct.h cvode/cvode_bandpre.h	cvode/cvode_impl.h cvode/cvode_spils.h cvode/cvode_bbdpre.h
CVODES	Libraries	libsundials_cvodes.*lib*	
	Header files	cvodes/cvodes.h cvodes/cvodes_direct.h cvodes/cvodes_bandpre.h	cvodes/cvodes_impl.h cvodes/cvodes_spils.h cvodes/cvodes_bbdpre.h
ARKODE	Libraries	libsundials_arkode.*lib*	libsundials_farkode.a
			continued on next page

continued from last page			
	Header files	arkode/arkode.h arkode/arkode_direct.h arkode/arkode_bandpre.h	arkode/arkode_impl.h arkode/arkode_spils.h arkode/arkode_bbdpre.h
IDA	Libraries	libsundials_ida.*lib*	libsundials_fida.a
	Header files	ida/ida.h ida/ida_direct.h ida/ida_bbdpre.h	ida/ida_impl.h ida/ida_spils.h
IDAS	Libraries	libsundials_idas.*lib*	
	Header files	idas/idas.h idas/idas_direct.h idas/idas_bbdpre.h	idas/idas_impl.h idas/idas_spils.h
KINSOL	Libraries	libsundials_kinsol.*lib*	libsundials_fkinsol.a
	Header files	kinsol/kinsol.h kinsol/kinsol_direct.h kinsol/kinsol_bbdpre.h	kinsol/kinsol_impl.h kinsol/kinsol_spils.h

Appendix B

CVODES 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 CVODES input constants

CVODES main solver module		
CV_ADAMS	1	Adams-Moulton linear multistep method.
CV_BDF	2	BDF linear multistep method.
CV_FUNCTIONAL	1	Nonlinear system solution through functional iterations.
CV_NEWTON	2	Nonlinear system solution through Newton iterations.
CV_NORMAL	1	Solver returns at specified output time.
CV_ONE_STEP	2	Solver returns after each successful step.
CV_SIMULTANEOUS	1	Simultaneous corrector forward sensitivity method.
CV_STAGGERED	2	Staggered corrector forward sensitivity method.
CV_STAGGERED1	3	Staggered (variant) corrector forward sensitivity method.
CV_CENTERED	1	Central difference quotient approximation (2^{nd} order) of the sensitivity RHS.
CV_FORWARD	2	Forward difference quotient approximation (1^{st} order) of the sensitivity RHS.
CVODES adjoint solver module		
CV_HERMITE	1	Use Hermite interpolation.
CV_POLYNOMIAL	2	Use variable-degree polynomial interpolation.
Iterative linear solver module		
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 CVODES output constants

CVODES **main solver module**		
CV_SUCCESS	0	Successful function return.
CV_TSTOP_RETURN	1	CVode succeeded by reaching the specified stopping point.
CV_ROOT_RETURN	2	CVode succeeded and found one or more roots.
CV_WARNING	99	CVode succeeded but an unusual situation occurred.
CV_TOO_MUCH_WORK	-1	The solver took mxstep internal steps but could not reach tout.
CV_TOO_MUCH_ACC	-2	The solver could not satisfy the accuracy demanded by the user for some internal step.
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_CONSTR_FAIL	-13	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 cvode_mem argument was NULL.
CV_ILL_INPUT	-22	One of the function inputs is illegal.
CV_NO_MALLOC	-23	The CVODE memory block was not allocated by a call to CVodeMalloc.
CV_BAD_K	-24	The derivative order k is larger than the order used.
CV_BAD_T	-25	The time t is outside the last step taken.
CV_BAD_DKY	-26	The output derivative vector is NULL.
CV_TOO_CLOSE	-27	The output and initial times are too close to each other.
CV_NO_QUAD	-30	Quadrature integration was not activated.

CV_QRHSFUNC_FAIL	-31	The quadrature right-hand side function failed in an unrecoverable manner.
CV_FIRST_QRHSFUNC_ERR	-32	The quadrature right-hand side function failed at the first call.
CV_REPTD_QRHSFUNC_ERR	-33	The quadrature ight-hand side function had repetead recoverable errors.
CV_UNREC_QRHSFUNC_ERR	-34	The quadrature right-hand side function had a recoverable error, but no recovery is possible.
CV_NO_SENS	-40	Forward sensitivity integration was not activated.
CV_SRHSFUNC_FAIL	-41	The sensitivity right-hand side function failed in an unrecoverable manner.
CV_FIRST_SRHSFUNC_ERR	-42	The sensitivity right-hand side function failed at the first call.
CV_REPTD_SRHSFUNC_ERR	-43	The sensitivity ight-hand side function had repetead recoverable errors.
CV_UNREC_SRHSFUNC_ERR	-44	The sensitivity right-hand side function had a recoverable error, but no recovery is possible.
CV_BAD_IS	-45	The sensitivity index is larger than the number of sensitivities computed.
CV_NO_QUADSENS	-50	Forward sensitivity integration was not activated.
CV_QSRHSFUNC_FAIL	-51	The sensitivity right-hand side function failed in an unrecoverable manner.
CV_FIRST_QSRHSFUNC_ERR	-52	The sensitivity right-hand side function failed at the first call.
CV_REPTD_QSRHSFUNC_ERR	-53	The sensitivity ight-hand side function had repetead recoverable errors.
CV_UNREC_QSRHSFUNC_ERR	-54	The sensitivity right-hand side function had a recoverable error, but no recovery is possible.

CVODES **adjoint solver module**

CV_NO_ADJ	-101	Adjoint module was not initialized.
CV_NO_FWD	-102	The forward integration was not yet performed.
CV_NO_BCK	-103	No backward problem was specified.
CV_BAD_TBO	-104	The final time for the adjoint problem is outside the interval over which the forward problem was solved.
CV_REIFWD_FAIL	-105	Reinitialization of the forward problem failed at the first checkpoint.
CV_FWD_FAIL	-106	An error occurred during the integration of the forward problem.
CV_GETY_BADT	-107	Wrong time in interpolation function.

CVDLS **linear solver modules**

CVDLS_SUCCESS	0	Successful function return.
CVDLS_MEM_NULL	-1	The cvode_mem argument was NULL.
CVDLS_LMEM_NULL	-2	The CVDLS linear solver has not been initialized.

CVDLS_ILL_INPUT	-3	The CVDLS solver is not compatible with the current NVECTOR module.
CVDLS_MEM_FAIL	-4	A memory allocation request failed.
CVDLS_JACFUNC_UNRECVR	-5	The Jacobian function failed in an unrecoverable manner.
CVDLS_JACFUNC_RECVR	-6	The Jacobian function had a recoverable error.
CVDLS_SUNMAT_FAIL	-7	An error occurred with the current SUNMATRIX module.
CVDLS_NO_ADJ	-101	The combined forward-backward problem has not been initialized.
CVDLS_LMEMB_NULL	-102	The linear solver was not initialized for the backward phase.

CVDIAG linear solver module

CVDIAG_SUCCESS	0	Successful function return.
CVDIAG_MEM_NULL	-1	The cvode_mem argument was NULL.
CVDIAG_LMEM_NULL	-2	The CVDIAG linear solver has not been initialized.
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.
CVDIAG_NO_ADJ	-101	The combined forward-backward problem has not been initialized.

CVSPILS linear solver modules

CVSPILS_SUCCESS	0	Successful function return.
CVSPILS_MEM_NULL	-1	The cvode_mem argument was NULL.
CVSPILS_LMEM_NULL	-2	The CVSPILS linear solver has not been initialized.
CVSPILS_ILL_INPUT	-3	The CVSPILS solver is not compatible with the current NVECTOR module, or an input value was illegal.
CVSPILS_MEM_FAIL	-4	A memory allocation request failed.
CVSPILS_PMEM_NULL	-5	The preconditioner module has not been initialized.
CVSPILS_SUNLS_FAIL	-6	An error occurred with the current SUNLINSOL module.
CVSPILS_NO_ADJ	-101	The combined forward-backward problem has not been initialized.
CVSPILS_LMEMB_NULL	-102	The linear solver was not initialized for the backward phase.

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] 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.

[4] P. N. Brown and A. C. Hindmarsh. Reduced Storage Matrix Methods in Stiff ODE Systems. *J. Appl. Math. & Comp.*, 31:49–91, 1989.

[5] 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.

[6] 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.

[7] 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.

[8] Y. Cao, S. Li, L. R. Petzold, and R. Serban. Adjoint Sensitivity Analysis for Differential-Algebraic Equations: The Adjoint DAE System and its Numerical Solution. *SIAM J. Sci. Comput.*, 24(3):1076–1089, 2003.

[9] M. Caracotsios and W. E. Stewart. Sensitivity Analysis of Initial Value Problems with Mixed ODEs and Algebraic Equations. *Computers and Chemical Engineering*, 9:359–365, 1985.

[10] S. D. Cohen and A. C. Hindmarsh. CVODE, a Stiff/Nonstiff ODE Solver in C. *Computers in Physics*, 10(2):138–143, 1996.

[11] 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.

[12] 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.

[13] W. F. Feehery, J. E. Tolsma, and P. I. Barton. Efficient Sensitivity Analysis of Large-Scale Differential-Algebraic Systems. *Applied Numer. Math.*, 25(1):41–54, 1997.

[14] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14:470–482, 1993.

[15] 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.

[16] K. L. Hiebert and L. F. Shampine. Implicitly Defined Output Points for Solutions of ODEs. Technical Report SAND80-0180, Sandia National Laboratories, February 1980.

[17] 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.

[18] A. C. Hindmarsh. Avoiding BDF Stability Barriers in the MOL Solution of Advection-Dominated Problems. *Appl. Num. Math.*, 17:311–318, 1995.

[19] A. C. Hindmarsh. The PVODE and IDA Algorithms. Technical Report UCRL-ID-141558, LLNL, December 2000.

[20] 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.

[21] A. C. Hindmarsh and R. Serban. User Documentation for CVODE v3.2.0. Technical Report UCRL-SM-208108, LLNL, 2018.

[22] A. C. Hindmarsh, R. Serban, and A. Collier. Example Programs for IDA v3.2.0. Technical Report UCRL-SM-208113, LLNL, 2018.

[23] A. C. Hindmarsh, R. Serban, and D. R. Reynolds. Example Programs for CVODE v3.2.0. Technical report, LLNL, 2018. UCRL-SM-208110.

[24] A. C. Hindmarsh and A. G. Taylor. PVODE and KINSOL: Parallel Software for Differential and Nonlinear Systems. Technical Report UCRL-ID-129739, LLNL, February 1998.

[25] 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.

[26] S. Li, L. R. Petzold, and W. Zhu. Sensitivity Analysis of Differential-Algebraic Equations: A Comparison of Methods on a Special Problem. *Applied Num. Math.*, 32:161–174, 2000.

[27] X. S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.

[28] T. Maly and L. R. Petzold. Numerical Methods and Software for Sensitivity Analysis of Differential-Algebraic Systems. *Applied Numerical Mathematics*, 20:57–79, 1997.

[29] D.B. Ozyurt and P.I. Barton. Cheap second order directional derivatives of stiff ODE embedded functionals. *SIAM J. of Sci. Comp.*, 26(5):1725–1743, 2005.

[30] 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.

[31] Daniel R. Reynolds. Example Programs for ARKODE v2.2.0. Technical report, Southern Methodist University, 2018.

[32] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14(2):461–469, 1993.

[33] 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.

[34] R. Serban and A. C. Hindmarsh. CVODES, the sensitivity-enabled ODE solver in SUNDIALS. In *Proceedings of the 5th International Conference on Multibody Systems, Nonlinear Dynamics and Control*, Long Beach, CA, 2005. ASME.

[35] R. Serban and A. C. Hindmarsh. Example Programs for CVODES v3.2.0. Technical Report UCRL-SM-208115, LLNL, 2017.

[36] H. A. Van Der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 13:631–644, 1992.

Index

Adams method, 13
adjoint sensitivity analysis
    checkpointing, 25
    implementation in CVODES, 26, 30
    mathematical background, 23–26
    quadrature evaluation, 143
    right-hand side evaluation, 142
    sensitivity-dependent quadrature evaluation, 144

BDF method, 13
BIG_REAL, 36, 160
booleantype, 36

CV_ADAMS, 41, 72, 129
CV_BAD_DKY, 57, 83, 102–104, 115, 116
CV_BAD_IS, 103, 104, 116
CV_BAD_ITASK, 133
CV_BAD_K, 57, 83, 103, 104, 115, 116
CV_BAD_T, 57, 83, 103, 104, 115, 116
CV_BAD_TBO, 129, 130
CV_BAD_TBOUT, 134
CV_BCKMEM_NULL, 134
CV_BDF, 41, 72, 129
CV_CENTERED, 104
CV_CONSTR_FAIL, 47
CV_CONV_FAILURE, 47, 128, 133
CV_ERR_FAILURE, 47, 128, 133
CV_FIRST_QRHSFUNC_ERR, 85
CV_FIRST_QRHSFUNC_FAIL, 81
CV_FIRST_QSRHSFUNC_ERR, 114, 120
CV_FIRST_RHSFUNC_ERR, 73
CV_FIRST_RHSFUNC_FAIL, 47
CV_FIRST_SRHSFUNC_ERR, 102, 110, 111
CV_FORWARD, 104
CV_FUNCTIONAL, 41, 53, 129
CV_FWD_FAIL, 134
CV_GETY_BADT, 139
CV_HERMITE, 126
CV_ILL_INPUT, 42, 47, 50–53, 56, 72, 83, 84, 98–101, 104, 105, 113, 114, 117, 118, 126, 128–131, 133–135, 140, 141
CV_LINIT_FAIL, 47
CV_LSETUP_FAIL, 47, 75, 89, 90, 128, 133, 145, 146, 153, 154
CV_LSOLVE_FAIL, 47, 128
CV_MEM_FAIL, 42, 80, 98–100, 126, 128, 129, 140
CV_MEM_NULL, 41–43, 47, 49–53, 56, 57, 59–66, 72, 80–85, 98–109, 129–131, 133, 134, 139–141
CV_NEWTON, 41, 53, 129
CV_NO_ADJ, 127–131, 133–135, 140, 141
CV_NO_BCK, 133
CV_NO_FWD, 133
CV_NO_MALLOC, 42, 43, 47, 72, 128–131
CV_NO_QUAD, 81, 83–85, 117, 140
CV_NO_QUADSENS, 114–119
CV_NO_SENS, 100–104, 106–109, 113, 114, 116
CV_NORMAL, 46, 124, 128, 133
CV_ONE_STEP, 46, 124, 128, 133
CV_POLYNOMIAL, 126
CV_QRHS_FAIL, 120
CV_QRHSFUNC_FAIL, 81, 85, 143, 144
CV_QSRHSFUNC_ERR, 114
CV_REIFWD_FAIL, 134
CV_REPTD_QRHSFUNC_ERR, 82
CV_REPTD_QSRHSFUNC_ERR, 114
CV_REPTD_RHSFUNC_ERR, 47
CV_REPTD_SRHSFUNC_ERR, 102
CV_RHSFUNC_FAIL, 47, 73, 142, 143
CV_ROOT_RETURN, 47, 128
CV_RTFUNC_FAIL, 47, 74
CV_SIMULTANEOUS, 30, 98, 99, 110
CV_SOLVE_FAIL, 133
CV_SRHSFUNC_FAIL, 102, 110, 111
CV_STAGGERED, 30, 98, 99, 110
CV_STAGGERED1, 30, 99, 111
CV_SUCCESS, 41–43, 46, 49–53, 56, 57, 59–66, 72, 80–85, 98–109, 113–119, 126–131, 133, 134, 139–141
CV_TOO_CLOSE, 47
CV_TOO_MUCH_ACC, 47, 128, 133
CV_TOO_MUCH_WORK, 47, 128, 133
CV_TSTOP_RETURN, 47, 128
CV_UNREC_QRHSFUNC_ERR, 85
CV_UNREC_QSRHSFUNC_ERR, 120

CV_UNREC_RHSFUNC_ERR, 47, 73, 82
CV_UNREC_SRHSFUNC_ERR, 102, 110, 111
CV_WARNING, 74
CVBANDPRE preconditioner
  description, 86
  optional output, 87–88
  usage, 86–87
  usage with adjoint module, 151–152
  user-callable functions, 87, 151–152
CVBandPrecGetNumRhsEvals, 88
CVBandPrecGetWorkSpace, 87
CVBandPrecInit, 87
CVBandPrecInitB, 151
CVBBDPRE preconditioner
  description, 88–89
  optional output, 92–93
  usage, 90–91
  usage with adjoint module, 152–154
  user-callable functions, 91–92, 152–153
  user-supplied functions, 89–90, 153–154
CVBBDPrecGetNumGfnEvals, 93
CVBBDPrecGetWorkSpace, 92
CVBBDPrecInit, 91
CVBBDPrecInitB, 152
CVBBDPrecReInit, 92
CVBBDPrecReInitB, 153
CVDENSE linear solver
  optional input, 135
CVDIAG linear solver
  Jacobian approximation used by, 45
  selection of, 45
CVDIAG linear solver interface
  memory requirements, 70
  optional output, 70–71
CVDiag, 39, 44, 45
CVDIAG_ILL_INPUT, 45
CVDIAG_LMEM_NULL, 71
CVDIAG_MEM_FAIL, 45
CVDIAG_MEM_NULL, 45, 71
CVDIAG_SUCCESS, 45, 71
CVDiagGetLastFlag, 71
CVDiagGetNumRhsEvals, 71
CVDiagGetReturnFlagName, 71
CVDiagGetWorkSpace, 70
CVDLS linear solver
  SUNLINSOL compatibility, 44
CVDLS linear solver interface
  Jacobian approximation used by, 54
  memory requirements, 66
  optional input, 54, 135–136
  optional output, 66–67
CVDLS_ILL_INPUT, 45, 132, 135, 136
CVDLS_JACFUNC_RECVR, 75, 145, 146
CVDLS_JACFUNC_UNRECVR, 75, 145, 146
CVDLS_LMEM_NULL, 54, 66, 67, 135, 136
CVDLS_MEM_FAIL, 45, 132
CVDLS_MEM_NULL, 45, 54, 66, 67, 132, 135, 136
CVDLS_NO_ADJ, 132, 135, 136
CVDLS_SUCCESS, 45, 54, 66, 67, 132, 135, 136
CVDlsGetLastFlag, 67
CVDlsGetNumJacEvals, 66
CVDlsGetNumRhsEvals, 67
CVDlsGetReturnFlagName, 67
CVDlsGetWorkSpace, 66
CVDlsJacFn, 75
CVDlsJacFnB, 144
CVDlsJacFnBS, 144
CVDlsSetJacFn, 54
CVDlsSetJacFnB, 135
CVDlsSetJacFnBS, 135
CVDlsSetLinearSolver, 39, 44, 75
CVDlsSetLinearSolverB, 132, 144
CVErrHandlerFn, 73
CVEwtFn, 74
CVODE, 1
CVode, 40, 46, 118
CVODE_MEM_FAIL, 113
CVODE_MEM_NULL, 113–119
CVodeAdjFree, 127
CVodeAdjInit, 124, 126
CVodeAdjReInit, 127
CVodeAdjSetNoSensi, 134
CVodeB, 125, 133
CVodeCreate, 41
CVodeCreateB, 124, 129
CVodeF, 124, 127
CVodeFree, 40, 42
CVodeGetActualInitStep, 62
CVodeGetAdjCheckPointsInfo, 139
CVodeGetAdjCVodeBmem, 138
CVodeGetAdjY, 138
CVodeGetB, 134
CVodeGetCurrentOrder, 61
CVodeGetCurrentStep, 62
CVodeGetCurrentTime, 62
CVodeGetDky, 57
CVodeGetErrWeights, 63
CVodeGetEstLocalErrors, 63
CVodeGetIntegratorStats, 64
CVodeGetLastOrder, 61
CVodeGetLastStep, 62
CVodeGetNonlinSolvStats, 65
CVodeGetNumErrTestFails, 61
CVodeGetNumGEvals, 66
CVodeGetNumLinSolvSetups, 61
CVodeGetNumNonlinSolvConvFails, 64
CVodeGetNumNonlinSolvIters, 64
CVodeGetNumRhsEvals, 60

CVodeGetNumRhsEvalsSEns, 106
CVodeGetNumStabLimOrderReds, 63
CVodeGetNumSteps, 60
CVodeGetQuad, 82, 141
CVodeGetQuadB, 125, 141
CVodeGetQuadDky, 82
CVodeGetQuadErrWeights, 84
CVodeGetQuadNumErrTestFails, 84
CVodeGetQuadNumRhsEvals, 84
CVodeGetQuadSens, 115
CVodeGetQuadSens1, 116
CVodeGetQuadSensDky, 115
CVodeGetQuadSensDky1, 116
CVodeGetQuadSensErrWeights, 119
CVodeGetQuadSensNumErrTestFails, 118
CVodeGetQuadSensNumRhsEvals, 118
CVodeGetQuadSensStats, 119
CVodeGetQuadStats, 85
CVodeGetReturnFlagName, 65
CVodeGetRootInfo, 65
CVodeGetSens, 97, 102
CVodeGetSens1, 97, 103
CVodeGetSensDky, 97, 102
CVodeGetSensDky1, 97, 103
CVodeGetSensErrWeights, 108
CVodeGetSensNonlinSolvStats, 109
CVodeGetSensNumErrTestFails, 107
CVodeGetSensNumLinSolvSetups, 107
CVodeGetSensNumNonlinSolvConvFails, 108
CVodeGetSensNumNonlinSolvIters, 108
CVodeGetSensNumRhsEvals, 106
CVodeGetSensStats, 107
CVodeGetStgrSensNumNonlinSolvConvFails, 109
CVodeGetStgrSensNumNonlinSolvIters, 109
CVodeGetTolScaleFactor, 63
CVodeGetWorkSpace, 59
CVodeInit, 41, 72
CVodeInitB, 124, 129
CVodeInitBS, 124, 130
CVodeQuadFree, 81
CVodeQuadInit, 80, 81
CVodeQuadInitB, 139
CVodeQuadInitBS, 140
CVodeQuadReInit, 81
CVodeQuadReInitB, 140
CVodeQuadSensEEtolerances, 118
CVodeQuadSensFree, 114
CVodeQuadSensInit, 113, 114
CVodeQuadSensReInit, 114
CVodeQuadSensSStolerances, 117
CVodeQuadSensSVtolerances, 117
CVodeQuadSStolerances, 83
CVodeQuadSVtolerances, 83
CVodeReInit, 72
CVodeReInitB, 130
CVodeRootInit, 46
CVODES
  brief description of, 1
  motivation for writing in C, 2
  package structure, 29
  relationship to CVODE, PVODE, 2
  relationship to VODE, VODPK, 1–2
CVODES linear solver interfaces, 30
  CVDIAG, 45
  CVDLS, 44, 131
  CVSPILS, 45
  CVSpilsCVSPILS, 132
  selecting one, 44
CVODES linear solvers
  header files, 37
  implementation details, 33
  NVECTOR compatibility, 35
  selecting one, 44
  usage with adjoint module, 131
cvodes.h, 37
cvodes/cvodes_diag.h, 38
cvodes/cvodes_direct.h, 37
cvodes/cvodes_spils.h, 37
CVodeSensEEtolerances, 101
CVodeSensFree, 100
CVodeSensInit, 97–99
CVodeSensInit1, 97–99, 110
CVodeSensReInit, 99
CVodeSensSStolerances, 101
CVodeSensSVtolerances, 101
CVodeSensToggleOff, 100
CVodeSetConstraints, 53
CVodeSetErrFile, 49
CVodeSetErrHandlerFn, 49
CVodeSetInitStep, 51
CVodeSetIterType, 53
CVodeSetMaxConvFails, 52
CVodeSetMaxErrTestFails, 52
CVodeSetMaxHnilWarns, 50
CVodeSetMaxNonlinIters, 52
CVodeSetMaxNumSteps, 50
CVodeSetMaxOrd, 50
CVodeSetMaxStep, 51
CVodeSetMinStep, 51
CVodeSetNoInactiveRootWarn, 56
CVodeSetNonlinConvCoef, 53
CVodeSetQuadErrCon, 83
CVodeSetQuadSensErrCon, 117
CVodeSetRootDirection, 56
CVodeSetSensDQMethod, 104
CVodeSetSensErrCon, 105
CVodeSetSensMaxNonlinIters, 105
CVodeSetSensParams, 104

CVodeSetStabLimDet, 50
CVodeSetStopTime, 52
CVodeSetUserData, 49
CVodeSStolerances, 42
CVodeSStolerancesB, 131
CVodeSVtolerances, 42
CVodeSVtolerancesB, 131
CVodeWFtolerances, 43
CVQuadRhsFn, 80, 85
CVQuadRhsFnB, 140, 143
CVQuadRhsFnBS, 140, 144
CVQuadSensRhsFn, 113, 119
CVRhsFn, 41, 73
CVRhsFnB, 129, 142
CVRhsFnBS, 130, 142
CVRootFn, 74
CVSensRhs1Fn, 99, 111
CVSensRhsFn, 98, 110
CVSPILS linear solver
  preconditioner setup function, 150
  preconditioner solve function, 149
  SUNLINSOL compatibility, 45
CVSPILS linear solver interface
  convergence test, 55
  Jacobian approximation used by, 54
  memory requirements, 67
  optional input, 54–56, 136–138
  optional output, 67–70
  preconditioner setup function, 54, 78
  preconditioner solve function, 54, 77
CVSPILS_ILL_INPUT, 45, 56, 87, 91, 132, 136–138, 152, 153
CVSPILS_LMEM_NULL, 55, 56, 68–70, 87, 91, 92, 136–138, 152, 153
CVSPILS_MEM_FAIL, 45, 87, 91, 132, 151–153
CVSPILS_MEM_NULL, 45, 55, 56, 68–70, 132, 136–138, 152, 153
CVSPILS_NO_ADJ, 132, 136–138
CVSPILS_PMEM_NULL, 87, 88, 92, 93, 153
CVSPILS_SUCCESS, 45, 55, 56, 68–70, 87, 88, 132, 136–138, 151–153
CVSPILS_SUNLS_FAIL, 45, 55
CVSpilsGetLastFlag, 70
CVSpilsGetNumConvFails, 68
CVSpilsGetNumJtimesEvals, 69
CVSpilsGetNumJTSetupEvals, 69
CVSpilsGetNumLinIters, 68
CVSpilsGetNumPrecEvals, 68
CVSpilsGetNumPrecSolves, 69
CVSpilsGetNumRhsEvals, 69
CVSpilsGetReturnFlagName, 70
CVSpilsGetWorkSpace, 68
CVSpilsJacTimesSetupFn, 77
CVSpilsJacTimesSetupFnB, 147
CVSpilsJacTimesVecFn, 76
CVSpilsJacTimesVecFnB, 146
CVSpilsJacTimesVecFnBS, 146
CVSpilsPrecSetupFn, 78
CVSpilsPrecSolveFn, 78
CVSpilsSetEpsLin, 56
CVSpilsSetEpsLinB, 138
CVSpilsSetJacTimes, 55
CVSpilsSetJacTimesB, 137
CVSpilsSetJacTimesBS, 137
CVSpilsSetLinearSolver, 39, 44, 45, 132
CVSpilsSetPreconditioner, 55
CVSpilsSetPrecSolveFnB, 136
CVSpilsSetPrecSolveFnBS, 136

eh_data, 74
error control
  order selection, 16–17
  sensitivity variables, 22
  step size selection, 16
error messages, 48
  redirecting, 48
  user-defined handler, 49, 73

forward sensitivity analysis
  absolute tolerance selection, 22
  correction strategies, 21–22, 29, 98, 100
  mathematical background, 20–23
  right hand side evaluation, 23
  right-hand side evaluation, 22, 110–111

getDevData(N_Vector v), 174, 176
getGlobalSize(N_Vector v), 174, 177
getHostData(N_Vector v), 174, 177
getMPIComm(N_Vector v), 174, 177
getSize(N_Vector v), 174, 177

half-bandwidths, 87, 91
header files, 37, 86, 90

itask, 40, **46**, 128
iter, 41, 53

Jacobian approximation function
  diagonal
    difference quotient, 45
  difference quotient, 54
  direct
    user-supplied (backward), 135
  Jacobian times vector
    difference quotient, 54
    user-supplied, 55
  Jacobian-vector product
    user-supplied, 76–77
    user-supplied (backward), 137, 146

Jacobian-vector setup, 77
user-supplied (backward), 147
user-supplied, 54, 75–76, 135
user-supplied (backward), 144, 145

lmm, 41, 72
LSODE, 1

maxord, 50, 72
memory requirements
CVBANDPRE preconditioner, 87
CVBBDPRE preconditioner, 92
CVDIAG linear solver interface, 70
CVDLS linear solver interface, 66
CVODES solver, 81, 99, 113
CVODES solver, 60
CVSPILS linear solver interface, 67

N_VCloneVectorArray, 156
N_VCloneVectorArray_OpenMP, 166
N_VCloneVectorArray_Parallel, 164
N_VCloneVectorArray_ParHyp, 171
N_VCloneVectorArray_Petsc, 172
N_VCloneVectorArray_Pthreads, 169
N_VCloneVectorArray_Serial, 161
N_VCloneVectorArrayEmpty, 156
N_VCloneVectorArrayEmpty_OpenMP, 167
N_VCloneVectorArrayEmpty_Parallel, 164
N_VCloneVectorArrayEmpty_ParHyp, 171
N_VCloneVectorArrayEmpty_Petsc, 172
N_VCloneVectorArrayEmpty_Pthreads, 169
N_VCloneVectorArrayEmpty_Serial, 161
N_VCopyFromDevice_Cuda, 175
N_VCopyFromDevice_Raja, 178
N_VCopyToDevice_Cuda, 175
N_VCopyToDevice_Raja, 178
N_VDestroyVectorArray, 156
N_VDestroyVectorArray_OpenMP, 167
N_VDestroyVectorArray_Parallel, 164
N_VDestroyVectorArray_ParHyp, 171
N_VDestroyVectorArray_Petsc, 173
N_VDestroyVectorArray_Pthreads, 169
N_VDestroyVectorArray_Serial, 161
N_Vector, 37, 155
N_VGetDeviceArrayPointer_Cuda, 175
N_VGetDeviceArrayPointer_Raja, 178
N_VGetHostArrayPointer_Cuda, 175
N_VGetHostArrayPointer_Raja, 178
N_VGetLength_Cuda, 175
N_VGetLength_OpenMP, 167
N_VGetLength_Parallel, 164
N_VGetLength_Pthreads, 169
N_VGetLength_Raja, 178
N_VGetLength_Serial, 162
N_VGetLocalLength_Parallel, 164
N_VGetVector_ParHyp, 171
N_VGetVector_Petsc, 172
N_VMake_Cuda, 175
N_VMake_OpenMP, 166
N_VMake_Parallel, 164
N_VMake_ParHyp, 170
N_VMake_Petsc, 172
N_VMake_Pthreads, 169
N_VMake_Raja, 177
N_VMake_Serial, 161
N_VNew_Cuda, 174, 175
N_VNew_OpenMP, 166
N_VNew_Parallel, 163
N_VNew_Pthreads, 169
N_VNew_Raja, 177
N_VNew_Serial, 161
N_VNewEmpty_Cuda, 175
N_VNewEmpty_OpenMP, 166
N_VNewEmpty_Parallel, 164
N_VNewEmpty_ParHyp, 170
N_VNewEmpty_Petsc, 172
N_VNewEmpty_Pthreads, 169
N_VNewEmpty_Raja, 177
N_VNewEmpty_Serial, 161
N_VPrint_Cuda, 175
N_VPrint_OpenMP, 167
N_VPrint_Parallel, 164
N_VPrint_ParHyp, 171
N_VPrint_Petsc, 173
N_VPrint_Pthreads, 169
N_VPrint_Raja, 178
N_VPrint_Serial, 162
N_VPrintFile_Cuda, 175
N_VPrintFile_OpenMP, 167
N_VPrintFile_Parallel, 164
N_VPrintFile_ParHyp, 171
N_VPrintFile_Petsc, 173
N_VPrintFile_Pthreads, 169
N_VPrintFile_Raja, 178
N_VPrintFile_Serial, 162
nonlinear system
definition, 13–14
Newton convergence test, 15
Newton iteration, 14–15
NV_COMM_P, 163
NV_CONTENT_OMP, 165
NV_CONTENT_P, 163
NV_CONTENT_PT, 168
NV_CONTENT_S, 160
NV_DATA_OMP, 166
NV_DATA_P, 163
NV_DATA_PT, 168
NV_DATA_S, 160
NV_GLOBLENGTH_P, 163

NV_Ith_OMP, 166
NV_Ith_P, 163
NV_Ith_PT, 168
NV_Ith_S, 161
NV_LENGTH_OMP, 166
NV_LENGTH_PT, 168
NV_LENGTH_S, 160
NV_LOCLENGTH_P, 163
NV_NUM_THREADS_OMP, 166
NV_NUM_THREADS_PT, 168
NV_OWN_DATA_OMP, 166
NV_OWN_DATA_P, 163
NV_OWN_DATA_PT, 168
NV_OWN_DATA_S, 160
NVECTOR module, 155

optional input
  backward solver, 135
  direct linear solver interface, 54, 135–136
  forward sensitivity, 104–105
  iterative linear solver, 54–56, 136–138
  quadrature integration, 83–84, 141
  rootfinding, 56
  sensitivity-dependent quadrature integration, 116–118
  solver, 48–54
optional output
  backward solver, 138
  band-block-diagonal preconditioner, 92–93
  banded preconditioner, 87–88
  diagonal linear solver interface, 70–71
  direct linear solver interface, 66–67
  forward sensitivity, 105–109
  interpolated quadratures, 82
  interpolated sensitivities, 102
  interpolated sensitivity-dep. quadratures, 115
  interpolated solution, 57
  iterative linear solver interface, 67–70
  quadrature integration, 84–85, 141
  sensitivity-dependent quadrature integration, 118–119
  solver, 59–65
  version, 57–59
output mode, 17, 46, 128, 133

partial error control
  explanation of CVODES behavior, 120
portability, 36
preconditioning
  advice on, 17, 33
  band-block diagonal, 88
  banded, 86
  setup and solve phases, 33
  user-supplied, 54–55, 77, 78, 136–137, 149, 150
PVODE, 2

quadrature integration, 20
  forward sensitivity analysis, 23

RCONST, 36
realtype, 36
reinitialization, 72, 130
right-hand side function, 73
  backward problem, 142
  forward sensitivity, 110–111
  quadrature backward problem, 143
  quadrature equations, 85
  sensitivity-dep. quadrature backward problem, 144
  sensitivity-dependent quadrature equations, 119
Rootfinding, 19, 40, 46

second-order sensitivity analysis, 26
  support in CVODES, 27
SM_COLS_B, 191
SM_COLS_D, 187
SM_COLUMN_B, 76, 191
SM_COLUMN_D, 75, 187
SM_COLUMN_ELEMENT_B, 76, 191
SM_COLUMNS_B, 191
SM_COLUMNS_D, 186
SM_COLUMNS_S, 197
SM_CONTENT_B, 191
SM_CONTENT_D, 186
SM_CONTENT_S, 195
SM_DATA_B, 191
SM_DATA_D, 187
SM_DATA_S, 197
SM_ELEMENT_B, 76, 191
SM_ELEMENT_D, 75, 187
SM_INDEXPTRS_S, 197
SM_INDEXVALS_S, 197
SM_LBAND_B, 191
SM_LDATA_B, 191
SM_LDATA_D, 186
SM_LDIM_B, 191
SM_NNZ_S, 76, 197
SM_NP_S, 197
SM_ROWS_B, 191
SM_ROWS_D, 186
SM_ROWS_S, 197
SM_SPARSETYPE_S, 197
SM_SUBAND_B, 191
SM_UBAND_B, 191
SMALL_REAL, 36
Stability limit detection, 18
step size bounds, 51
SUNBandLinearSolver, 213

SUNBandMatrix, 192
SUNBandMatrix_Cols, 193
SUNBandMatrix_Column, 193
SUNBandMatrix_Columns, 192
SUNBandMatrix_Data, 193
SUNBandMatrix_LDim, 193
SUNBandMatrix_LowerBandwidth, 192
SUNBandMatrix_Print, 192
SUNBandMatrix_Rows, 192
SUNBandMatrix_StoredUpperBandwidth, 192
SUNBandMatrix_UpperBandwidth, 192
SUNDenseLinearSolver, 211
SUNDenseMatrix, 187
SUNDenseMatrix_Cols, 188
SUNDenseMatrix_Column, 188
SUNDenseMatrix_Columns, 188
SUNDenseMatrix_Data, 188
SUNDenseMatrix_LData, 188
SUNDenseMatrix_Print, 187
SUNDenseMatrix_Rows, 188
sundials_nvector.h, 37
sundials_types.h, 36, 37
SUNDIALSGetVersion, 59
SUNDIALSGetVersionNumber, 59
sunindextype, 36
SUNKLU, 218
SUNKLUReInit, 218
SUNKLUSetOrdering, 219
SUNLapackBand, 216
SUNLapackDense, 214
SUNLinearSolver, 203, 204
SUNLinearSolver module, 203
SUNLINEARSOLVER_DIRECT, 205
SUNLINEARSOLVER_ITERATIVE, 205
sunlinsol/sunlinsol_band.h, 37
sunlinsol/sunlinsol_dense.h, 37
sunlinsol/sunlinsol_klu.h, 37
sunlinsol/sunlinsol_lapackband.h, 37
sunlinsol/sunlinsol_lapackdense.h, 37
sunlinsol/sunlinsol_pcg.h, 38
sunlinsol/sunlinsol_spbcgs.h, 37
sunlinsol/sunlinsol_spfgmr.h, 37
sunlinsol/sunlinsol_spgmr.h, 37
sunlinsol/sunlinsol_sptfqmr.h, 37
sunlinsol/sunlinsol_superlumt.h, 37
SUNLinSolFree, 40
SUNMatDestroy, 40
SUNMatrix, 183
SUNMatrix module, 183
SUNPCG, 237, 238
SUNPCGSetMaxl, 238
SUNPCGSetPrecType, 237
SUNSparseFromBandMatrix, 198
SUNSparseFromDenseMatrix, 197
SUNSparseMatrix, 197
SUNSparseMatrix_Columns, 198
SUNSparseMatrix_Data, 199
SUNSparseMatrix_IndexPointers, 199
SUNSparseMatrix_IndexValues, 199
SUNSparseMatrix_NNZ, 76, 199
SUNSparseMatrix_NP, 199
SUNSparseMatrix_Print, 198
SUNSparseMatrix_Realloc, 198
SUNSparseMatrix_Reallocate, 198
SUNSparseMatrix_Rows, 198
SUNSparseMatrix_SparseType, 199
SUNSPBCGS, 231
SUNSPBCGSSetMaxl, 231
SUNSPBCGSSetPrecType, 231
SUNSPFGMR, 228, 229
SUNSPFGMRSetGSType, 228
SUNSPFGMRSetMaxRestarts, 228
SUNSPFGMRSetPrecType, 228
SUNSPGMR, 224, 225
SUNSPGMRSetGSType, 225
SUNSPGMRSetMaxRestarts, 225
SUNSPGMRSetPrecType, 225
SUNSPTFQMR, 234
SUNSPTFQMRSetMaxl, 234
SUNSPTFQMRSetPrecType, 234
SUNSuperLUMT, 221
SUNSuperLUMTSetOrdering, 221, 222

tolerances, 14, 43, 74, 83, 84, 117

UNIT_ROUNDOFF, 36
User main program
  Adjoint sensitivity analysis, 123
  CVBANDPRE usage, 86
  CVBBDPRE usage, 90
  forward sensitivity analysis, 95
  integration of quadratures, 79
  integration of sensitivitiy-dependent quadratures, 111
  IVP solution, 38
user_data, 49, 73–75, 85, 89, 90, 110, 111, 120
user_dataB, 153, 154

VODE, 1
VODPK, 1

weighted root-mean-square norm, 14