User Documentation for CVODE v2.9.0 (SUNDIALS v2.7.0)

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

September 12, 2017

UCRL-SM-208108

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

Contents

List of Tables vii

List of Figures ix

1 Introduction 1
- 1.1 Historical Background . . . 1
- 1.2 Changes from previous versions . . . 2
- 1.3 Reading this User Guide . . . 4
- 1.4 SUNDIALS Release License . . . 5
  - 1.4.1 Copyright Notices . . . 6
    - 1.4.1.1 SUNDIALS Copyright . . . 6
    - 1.4.1.2 ARKode Copyright . . . 6
  - 1.4.2 BSD License . . . 6

2 Mathematical Considerations 9
- 2.1 IVP solution . . . 9
- 2.2 Preconditioning . . . 13
- 2.3 BDF stability limit detection . . . 13
- 2.4 Rootfinding . . . 14

3 Code Organization 17
- 3.1 SUNDIALS organization . . . 17
- 3.2 CVODE organization . . . 17

4 Using CVODE for C Applications 21
- 4.1 Access to library and header files . . . 21
- 4.2 Data Types . . . 22
- 4.3 Header files . . . 22
- 4.4 A skeleton of the user's main program . . . 23
- 4.5 User-callable functions . . . 25
  - 4.5.1 CVODE initialization and deallocation functions . . . 26
  - 4.5.2 CVODE tolerance specification functions . . . 26
  - 4.5.3 Linear solver specification functions . . . 28
  - 4.5.4 Rootfinding initialization function . . . 33
  - 4.5.5 CVODE solver function . . . 34
  - 4.5.6 Optional input functions . . . 35
    - 4.5.6.1 Main solver optional input functions . . . 35
    - 4.5.6.2 Dense/band direct linear solvers optional input functions . . . 41
    - 4.5.6.3 Sparse direct linear solvers optional input functions . . . 42
    - 4.5.6.4 Iterative linear solvers optional input functions . . . 44
    - 4.5.6.5 Rootfinding optional input functions . . . 47
  - 4.5.7 Interpolated output function . . . 47
  - 4.5.8 Optional output functions . . . 48

4.5.8.1 Main solver optional output functions . . . 48
4.5.8.2 Rootfinding optional output functions . . . 55
4.5.8.3 Dense/band direct linear solvers optional output functions . . . 56
4.5.8.4 Diagonal linear solver optional output functions . . . 57
4.5.8.5 Sparse direct linear solvers optional output functions . . . 59
4.5.8.6 Iterative linear solvers optional output functions . . . 60
4.5.9 CVODE reinitialization function . . . 62
4.6 User-supplied functions . . . 63
4.6.1 ODE right-hand side . . . 63
4.6.2 Error message handler function . . . 64
4.6.3 Error weight function . . . 65
4.6.4 Rootfinding function . . . 65
4.6.5 Jacobian information (direct method with dense Jacobian) . . . 65
4.6.6 Jacobian information (direct method with banded Jacobian) . . . 67
4.6.7 Jacobian information (direct method with sparse Jacobian) . . . 68
4.6.8 Jacobian information (matrix-vector product) . . . 69
4.6.9 Preconditioning (linear system solution) . . . 69
4.6.10 Preconditioning (Jacobian data) . . . 70
4.7 Preconditioner modules . . . 71
4.7.1 A serial banded preconditioner module . . . 71
4.7.2 A parallel band-block-diagonal preconditioner module . . . 73

5 FCVODE, an Interface Module for FORTRAN Applications 79
5.1 Important note on portability . . . 79
5.2 Fortran Data Types . . . 79
5.3 FCVODE routines . . . 80
5.4 Usage of the FCVODE interface module . . . 81
5.5 FCVODE optional input and output . . . 90
5.6 Usage of the FCVROOT interface to rootfinding . . . 92
5.7 Usage of the FCVBP interface to CVBANDPRE . . . 93
5.8 Usage of the FCVBBD interface to CVBBDPRE . . . 94

6 Description of the NVECTOR module 97
6.1 The NVECTOR_SERIAL implementation . . . 102
6.2 The NVECTOR_PARALLEL implementation . . . 104
6.3 The NVECTOR_OPENMP implementation . . . 107
6.4 The NVECTOR_PTHREADS implementation . . . 109
6.5 The NVECTOR_PARHYP implementation . . . 111
6.6 The NVECTOR_PETSC implementation . . . 113
6.7 NVECTOR Examples . . . 114
6.8 NVECTOR functions used by CVODE . . . 116

7 Description of the SUNMatrix module 119
7.1 The SUNMatrix_Dense implementation . . . 121
7.2 The SUNMatrix_Band implementation . . . 124
7.3 The SUNMatrix_Diagonal implementation . . . 128
7.4 The SUNMatrix_Sparse implementation . . . 130
7.5 SUNMatrix Examples . . . 136
7.6 SUNMatrix functions used by CVODE . . . 136

8 Description of the SUNLinearSolver module **139**
8.1 The SUNLinearSolver_Dense implementation . . . 145
8.2 The SUNLinearSolver_Band implementation . . . 146
8.3 The SUNLinearSolver_Diagonal implementation . . . 148
8.4 The SUNLinearSolver_LapackDense implementation . . . 149
8.5 The SUNLinearSolver_LapackBand implementation . . . 150
8.6 The SUNLinearSolver_KLU implementation . . . 152
8.7 The SUNLinearSolver_SuperLUMT implementation . . . 155
8.8 The SUNLinearSolver_SPGMR implementation . . . 155
8.9 The SUNLinearSolver_SPFGMR implementation . . . 155
8.10 The SUNLinearSolver_SPBCGS implementation . . . 155
8.11 The SUNLinearSolver_SPTFQMR implementation . . . 155
8.12 The SUNLinearSolver_PCG implementation . . . 155
8.13 SUNLinearSolver Examples . . . 155
8.14 SUNLinearSolver functions used by CVODE . . . 156

9 Providing Alternate Linear Solver Modules **159**
9.1 Initialization function . . . 160
9.2 Setup function . . . 160
9.3 Solve function . . . 161
9.4 Memory deallocation function . . . 162

10 General Use Linear Solver Components in SUNDIALS **163**
10.1 The DLS modules: DENSE and BAND . . . 164
10.1.1 Type DlsMat . . . 164
10.1.2 Accessor macros for the DLS modules . . . 167
10.1.3 Functions in the DENSE module . . . 167
10.1.4 Functions in the BAND module . . . 170
10.2 The SLS module . . . 171
10.2.1 Type SlsMat . . . 172
10.2.2 Functions in the SLS module . . . 175
10.2.3 The KLU solver . . . 176
10.2.4 The SUPERLUMT solver . . . 176
10.3 The SPILS modules: SPGMR, SPFGMR, SPBCG, and SPTFQMR . . . 176
10.3.1 The SPGMR module . . . 176
10.3.2 The SPFGMR module . . . 177
10.3.3 The SPBCG module . . . 178
10.3.4 The SPTFQMR module . . . 178

A SUNDIALS Package Installation Procedure **179**
A.1 CMake-based installation . . . 180
A.1.1 Configuring, building, and installing on Unix-like systems . . . 180
A.1.2 Configuration options (Unix/Linux) . . . 182
A.1.3 Configuration examples . . . 185
A.1.4 Working with external Libraries . . . 186
A.2 Building and Running Examples . . . 187
A.3 Configuring, building, and installing on Windows . . . 187
A.4 Installed libraries and exported header files . . . 188

B CVODE Constants **191**
B.1 CVODE input constants . . . 191
B.2 CVODE output constants . . . 191

Bibliography **197**

Index **199**

List of Tables

4.1 SUNDIALS linear solver interfaces and vector implementations that can be used for each. 25
4.2 Optional inputs for CVODE, CVDLS, CVSLS, and CVSPILS 36
4.3 Optional outputs from CVODE, CVDLS, CVDIAG, CVSLS, and CVSPILS 49

5.1 Keys for setting FCVODE optional inputs 90
5.2 Description of the FCVODE optional output arrays `IOUT` and `ROUT` 91

6.1 Vector Identifications associated with vector kernels supplied with SUNDIALS. 99
6.2 Description of the NVECTOR operations 99
6.3 List of vector functions usage by CVODE code modules 117

7.1 Identifiers associated with matrix kernels supplied with SUNDIALS. 120
7.2 Description of the `SUNMatrix` operations 120
7.3 List of matrix functions usage by CVODE code modules 137

8.1 Identifiers associated with linear solver kernels supplied with SUNDIALS. 141
8.2 Description of the `SUNLinearSolver` operations 141
8.3 Description of the `SUNLinearSolver` error codes 144
8.4 List of linear solver functions usage by CVODE code modules 157

A.1 SUNDIALS libraries and header files 189
A.2 SUNDIALS libraries and header files (cont.) 190

List of Figures

3.1 Organization of the SUNDIALS suite 18
3.2 Overall structure diagram of the CVODE package 19

7.1 Diagram of the storage for a banded matrix of type `DlsMat` 125
7.2 Diagram of the storage for a compressed-sparse-column matrix 132

10.1 Diagram of the storage for a banded matrix of type `DlsMat` 166
10.2 Diagram of the storage for a compressed-sparse-column matrix of type `SlsMat` 174

A.1 Initial *ccmake* configuration screen 181
A.2 Changing the *instdir* 182

Chapter 1

Introduction

CVODE is part of a software family called SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic equation Solvers [18]. This suite consists of CVODE, ARKODE, KINSOL, and IDA, and variants of these with sensitivity analysis capabilities.

1.1 Historical Background

FORTRAN solvers for ODE initial value problems are widespread and heavily used. Two solvers that have been written at LLNL in the past are VODE [3] and VODPK [5]. VODE is a general purpose solver that includes methods for 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 [24]. 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 contains three Krylov methods that can be used in conjuction with Newton iteration: the GMRES (Generalized Minimal RESidual) [26], Bi-CGStab (Bi-Conjugate Gradient Stabilized) [27], and TFQMR (Transpose-Free Quasi-Minimal Residual) linear iterative methods [13]. 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 three Krylov methods in CVODE, we recommend GMRES as the best overall choice. However, users are encouraged to compare all three, especially if encountering convergence failures with GMRES. Bi-CGFStab and TFQMR have an advantage in storage requirements, in that the number of workspace vectors they require is fixed, while that number for GMRES depends on the desired Krylov subspace size.

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 [8], the parallel variant of CVODE.

Recently, the functionality of CVODE and PVODE has been combined into one single code, simply called CVODE. Development of the new version of CVODE was concurrent with a redesign of the vector operations module across the SUNDIALS suite. The key feature of the NVECTOR module is that it is written in terms of abstract vector operations with the actual vector kernels attached by a particular

implementation (such as serial or parallel) of NVECTOR. This allows writing the SUNDIALS solvers in a manner independent of the actual NVECTOR implementation (which can be user-supplied), as well as allowing more than one NVECTOR module linked into an executable file. SUNDIALS (and thus CVODE) is supplied with six different NVECTOR implementations: serial, MPI-parallel, and both openMP and Pthreads thread-parallel NVECTOR implementations, a Hypre parallel implementation, and a PetSC implementation.

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

1.2 Changes from previous versions

Changes in v2.9.0

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

Each NVECTOR module now includes a function, `N_VGetVectorID`, that returns the NVECTOR module name.

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

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

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

The Krylov linear solver Bi-CGstab was enhanced by removing a redundant dot product. Various additions and corrections were made to the interfaces to the sparse solvers KLU and SuperLU_MT, including support for CSR format when using KLU.

New examples were added for use of the openMP vector and for use of sparse direct solvers from Fortran.

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

Changes in v2.8.0

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

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

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

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

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

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

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

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

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

In all FCVODE examples, integer declarations were revised so that those which must match a C type long int are declared INTEGER*8, and a comment was added about the type match. All other integer declarations are just INTEGER. Corresponding minor corrections were made to the user guide.

Two new NVECTOR modules have been added for thread-parallel computing environments — one for openMP, denoted NVECTOR_OPENMP, and one for Pthreads, denoted NVECTOR_PTHREADS.

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

Changes in v2.7.0

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

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

Changes in v2.6.0

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

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

Changes in v2.5.0

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

The functions in the generic dense linear solver (`sundials_dense` and `sundials_smalldense`) were modified to work for rectangular $m \times n$ matrices ($m \leq n$), 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 (SPBCG) and Scaled Preconditioned Transpose-Free Quasi-Minimal Residual (SPTFQMR) linear solver modules, respectively (for details see Chapter 4). Corresponding additions were made to the FORTRAN interface module FCVODE. At the same time, function type names for Scaled Preconditioned Iterative Linear Solvers were added for the user-supplied Jacobian-times-vector and preconditioner setup and solve functions.

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

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

Changes in v2.3.0

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

Changes in v2.2.1

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

Changes in v2.2.0

The major changes from the previous version involve a redesign of the user interface across the entire SUNDIALS suite. We have eliminated the mechanism of providing optional inputs and extracting optional statistics from the solver through the `iopt` and `ropt` arrays. Instead, CVODE now provides a set of routines (with prefix `CVodeSet`) to change the default values for various quantities controlling the solver and a set of extraction routines (with prefix `CVodeGet`) to extract statistics after return from the main solver routine. Similarly, each linear solver module provides its own set of `Set`- and `Get`-type routines. For more details see §4.5.6 and §4.5.8.

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

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

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

1.3 Reading this User Guide

This user guide is a combination of general usage instructions and specific example programs. We expect that some readers will want to concentrate on the general instructions, while others will refer

mostly to the examples, and the organization is intended to accommodate both styles.

There are different possible levels of usage of CVODE. The most casual user, with a small IVP problem only, can get by with reading §2.1, then Chapter 4 through §4.5.5 only, and looking at examples in [19]. In a different direction, a more expert user with an IVP problem may want to (a) use a package preconditioner (§4.7), (b) supply his/her own Jacobian or preconditioner routines (§4.6), (c) do multiple runs of problems of the same size (§4.5.9), (d) supply a new NVECTOR module (Chapter 6), or even (e) supply a different linear solver module (§3.2 and Chapter 9).

The structure of this document is as follows:

- In Chapter 2, we give short descriptions of the numerical methods implemented by CVODE for the solution of initial value problems for systems of ODEs, and continue with short descriptions of preconditioning (§2.2), stability limit detection (§2.3), and rootfinding (§2.4).
- The following chapter describes the structure of the SUNDIALS suite of solvers (§3.1) and the software organization of the CVODE solver (§3.2).
- Chapter 4 is the main usage document for CVODE for C applications. It includes a complete description of the user interface for the integration of ODE initial value problems.
- In Chapter 5, we describe FCVODE, an interface module for the use of CVODE with FORTRAN applications.
- Chapter 6 gives a brief overview of the generic NVECTOR module shared among the various components of SUNDIALS, and details on the NVECTOR implementations provided with SUNDIALS: a serial implementation (§6.1), a distributed memory parallel implementation based on MPI (§6.2), and two thread-parallel implementations based on openMP (§6.3) and Pthreads (§6.4), respectively.
- Chapter 9 describes the interfaces to the linear solver modules, so that a user can provide his/her own such module.
- Chapter 10 describes in detail the generic linear solvers shared by all SUNDIALS solvers.
- Finally, in the appendices, we provide detailed instructions for the installation of CVODE, within the structure of SUNDIALS (Appendix A), as well as a list of all the constants used for input to and output from CVODE functions (Appendix B).

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

Acknowledgments. We wish to acknowledge the contributions to previous versions of the CVODE and PVODE codes and their user guides by Scott D. Cohen [9] and George D. Byrne [7].

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

1.4.1.2 ARKode Copyright

1.4.2 BSD License

2. Neither the United States Government nor Lawrence Livermore National Security, LLC nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights.

3. Also, reference herein to any specific commercial products, process, or services by trade name, trademark, manufacturer or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Chapter 2

Mathematical Considerations

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

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

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

2.1 IVP solution

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

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

Here the y^n are computed approximations to $y(t_n)$, and $h_n = t_n - t_{n-1}$ is the step size. The user of CVODE must choose appropriately one of two multistep methods. For nonstiff problems, CVODE includes the Adams-Moulton formulas , characterized by $K_1 = 1$ and $K_2 = q$ above, where the order q varies between 1 and 12. For stiff problems, CVODE includes the Backward Differentiation Formulas (BDFs) in so-called fixed-leading coefficient 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 [22].

For either choice of formula, the nonlinear system

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

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, CVODE 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

$$M[y^{n(m+1)} - y^{n(m)}] = -G(y^{n(m)})\,, \tag{2.4}$$

in which

$$M \approx I - \gamma J\,, \quad J = \partial f/\partial y\,, \quad \text{and} \quad \gamma = h_n \beta_{n,0}\,. \tag{2.5}$$

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, CVODE provides several choices, including the option of an user-supplied linear solver module. The linear solver modules distributed with SUNDIALS are organized in three families, a *direct* family comprising direct linear solvers for dense or banded matrices, a *sparse* family comprising direct linear solvers for matrices stored in compressed-sparse-column format, and a *spils* family comprising scaled preconditioned iterative (Krylov) linear solvers. In addition, CVODE also provides a linear solver module which only uses a diagonal approximation of the Jacobian matrix. 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 [23, 12, 2] (serial or threaded vector modules only) [Note that users will need to download and install the KLU or SuperLU_MT packages independent of CVODE],
- a diagonal approximate Jacobian solver,
- SPGMR, a scaled preconditioned GMRES (Generalized Minimal Residual method) solver without restarts,
- SPBCG, a scaled preconditioned Bi-CGStab (Bi-Conjugate Gradient Stable method) solver, or
- SPTFQMR, a scaled preconditioned TFQMR (Transpose-Free Quasi-Minimal Residual method) solver.

For large stiff systems, where direct methods are not feasible, the combination of a BDF integrator and any of the preconditioned Krylov methods (SPGMR, SPBCG, or SPTFQMR) 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]. Note that the direct linear solvers (dense, band and sparse) can only be used with the serial and threaded vector representations.

In the process of controlling errors at various levels, CVODE uses a weighted root-mean-square norm, denoted $\|\cdot\|_{\mathrm{WRMS}}$, 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

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

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

In the 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 (2.3), we want to ensure that the iteration error $y^n - y^{n(m)}$ is small relative to ϵ, specifically that it is less than 0.1ϵ. (The safety factor 0.1 can be changed by the user.) For this, we also estimate the linear convergence rate constant R as follows. We initialize R to 1, and reset $R = 1$ when M or P is updated. After computing a correction $\delta_m = y^{n(m)} - y^{n(m-1)}$, we update R if $m > 1$ as

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

Now we use the estimate

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

Therefore the convergence (stopping) test is

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

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

When 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 (2.6). 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 compressed-sparse-column format.

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

$$Jv = [f(t, y + \sigma v) - f(t, y)]/\sigma \,. \tag{2.7}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and

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

The new order and step size are then set according to

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

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

The various algorithmic features of CVODE described above, as inherited from the solvers VODE and VODPK, are documented in [3, 5, 17]. They are also summarized in [18].

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

2.2 Preconditioning

When using a Newton method to solve the nonlinear system (2.3), CVODE makes repeated use of a linear solver to solve linear systems of the form $Mx = -r$, where x is a correction vector and r is a residual vector. If this linear system solve is done with one of the scaled preconditioned iterative linear solvers, 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).

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

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

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

When the BDF option is selected, CVODE 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 in 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 [15]. The algorithm supplements (but differs greatly from) the existing algorithms in CVODE for choosing step size and order based on estimated local truncation errors. It works directly with history data that is readily available in CVODE. If it concludes that the step size is in fact stability-limited, it dictates a reduction in the method order, regardless of the outcome of the error-based algorithm. The STALD algorithm has been tested in combination with the VODE solver on linear advection-dominated advection-diffusion problems [16], where it works well. The implementation in CVODE has been successfully tested on linear and nonlinear advection-diffusion problems, among others.

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

2.4 Rootfinding

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

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

The basic scheme used is to check for sign changes of any $g_i(t)$ over each time step taken, and then (when a sign change is found) to home in on the root (or roots) with a modified secant method [14].

In addition, each time g is computed, CVODE checks to see if $g_i(t) = 0$ exactly, and if so it reports this as a root. However, if an exact zero of any g_i is found at a point t, CVODE computes g at $t + \delta$ for a small increment δ, slightly further in the direction of integration, and if any $g_i(t + \delta) = 0$ also, CVODE stops and reports an error. This way, each time CVODE takes a time step, it is guaranteed that the values of all g_i are nonzero at some past value of t, beyond which a search for roots is to be done.

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

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

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

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

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

where α a weight parameter. On the first two passes through the loop, α is set to 1, making t_{mid} the secant method value. Thereafter, α is reset according to the side of the subinterval (low vs high, i.e. toward t_{lo} vs toward t_{hi}) in which the sign change was found in the previous two passes. If the two sides were opposite, α is set to 1. If the two sides were the same, α is halved (if on the low side) or doubled (if on the high side). The value of t_{mid} is closer to t_{lo} when $\alpha < 1$ and closer to t_{hi} when $\alpha > 1$. If the above value of t_{mid} is within $\tau/2$ of t_{lo} or t_{hi}, it is adjusted inward, such that its fractional distance from the endpoint (relative to the interval size) is between .1 and .5 (.5 being the midpoint), and the actual distance from the endpoint is at least $\tau/2$.

Chapter 3

Code Organization

3.1 SUNDIALS organization

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

The various solvers of this family share many subordinate modules. For this reason, it is organized as a family, with a directory structure that exploits that sharing (see Fig. 3.1). 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 $M dy/dt = f(t, y)$ based on additive Runge-Kutta methods;
- IDA, a solver for differential-algebraic systems $F(t, y, \dot{y}) = 0$ based on BDF methods;
- IDAS, a solver for differential-algebraic systems with sensitivity analysis capabilities;
- KINSOL, a solver for nonlinear algebraic systems $F(u) = 0$.

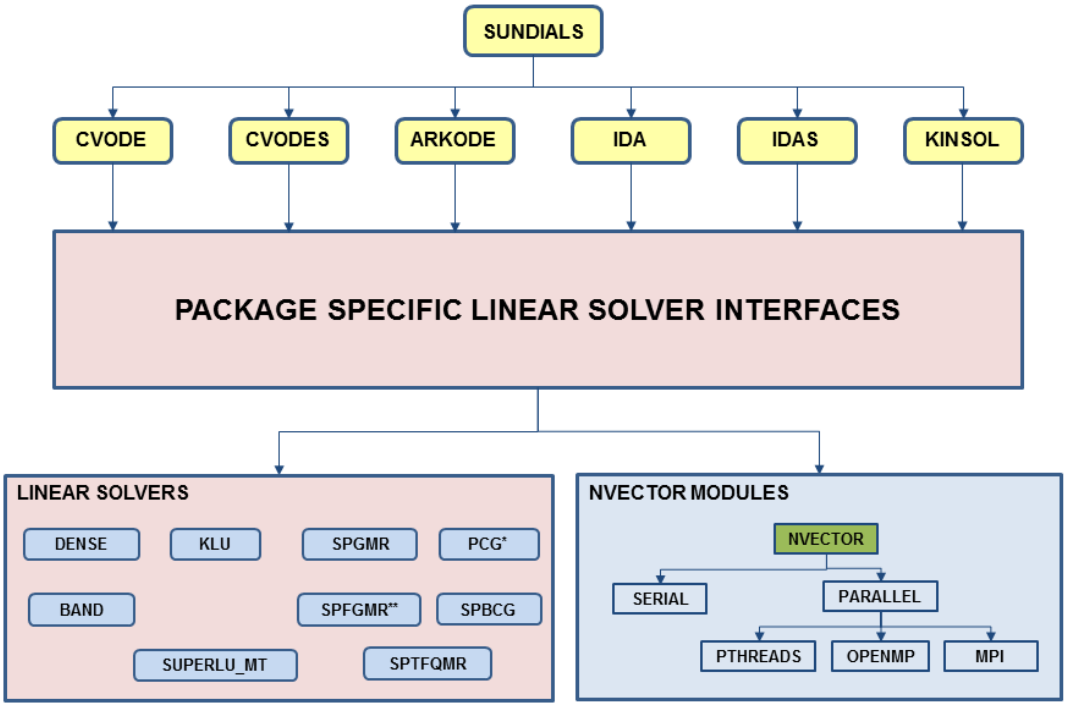
3.2 CVODE organization

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

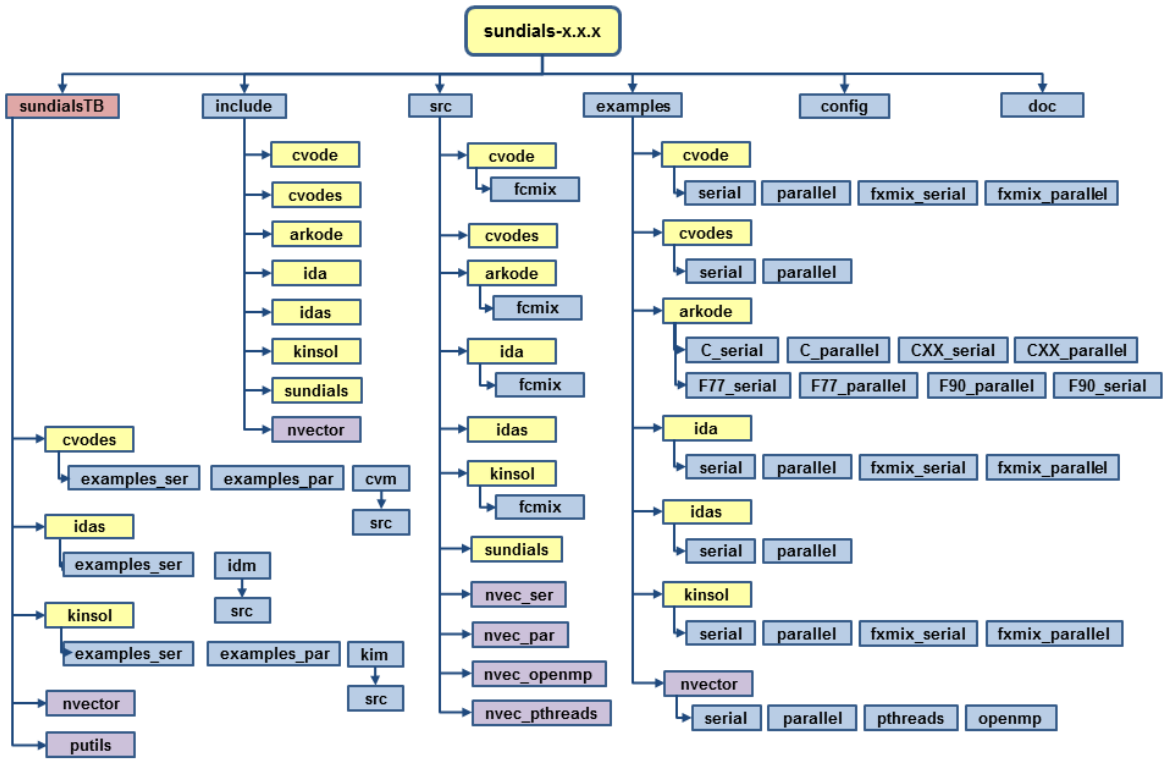
The overall organization of the CVODE package is shown in Figure 3.2. The central integration module, implemented in the files `cvode.h`, `cvode_impl.h`, and `cvode.c`, deals with the evaluation of integration coefficients, the functional or Newton iteration process, estimation of local error, selection 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 modules is specified, and is then invoked as needed during the integration.

At present, the package includes the following eight CVODE linear algebra modules, organized into two families. The *direct* family of linear solvers provides solvers for the direct solution of linear systems with dense, banded, or sparse matrices, and includes:

- CVDENSE: LU factorization and backsolving with dense matrices (using either an internal implementation or Blas/Lapack);



(a) High-level diagram (note that none of the Lapack-based linear solver modules are represented.)
* only applies to ARKODE
** only applies to ARKODE and KINSOL



(b) Directory structure of the source tree

Figure 3.1: Organization of the SUNDIALS suite

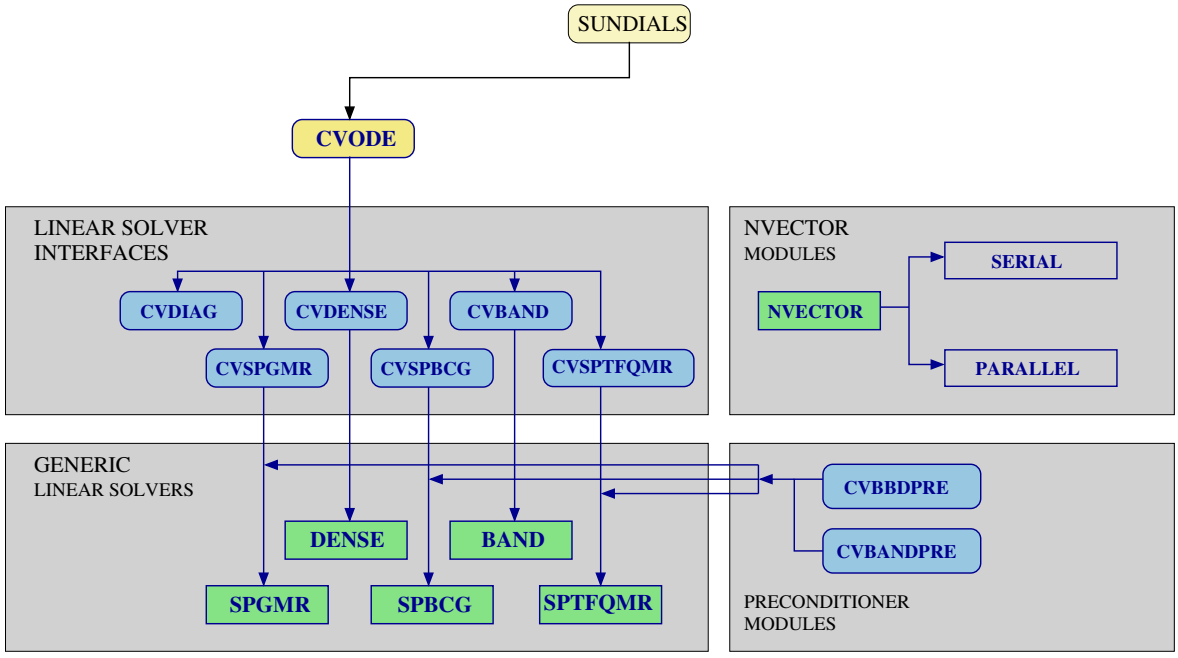


Figure 3.2: Overall structure diagram of the CVODE package. Modules specific to CVODE are distinguished by rounded boxes, while generic solver and auxiliary modules are in rectangular boxes. Note that the direct linear solvers using Lapack implementations are not explicitly represented. Note also that the KLU and SuperLU_MT support is through interfaces to packages. Users will need to download and compile those packages independently.

- CVBAND: LU factorization and backsolving with banded matrices (using either an internal implementation or Blas/Lapack);
- CVKLU: LU factorization and backsolving with compressed-sparse-column (CSC) matrices using the KLU linear solver library [11, 1] (KLU to be downloaded and compiled by user independent of CVODE);
- CVSUPERLUMT: LU factorization and backsolving with compressed-sparse-column (CSC) matrices using the threaded SuperLU_MT linear solver library [23, 12, 2] (SuperLU_MT to be downloaded and compiled by user independent of CVODE).

The *spils* family of linear solvers provides scaled preconditioned iterative linear solvers and includes:

- CVSPGMR: scaled preconditioned GMRES method;
- CVSPBCG: scaled preconditioned Bi-CGStab method;
- CVSPTFQMR: scaled preconditioned TFQMR method.

Additionally, CVODE includes:

- CVDIAG: an internally generated diagonal approximation to the Jacobian;

The set of linear solver modules distributed with CVODE is intended to be expanded in the future as new algorithms are developed. Note that users wishing to employ KLU or SuperLU_MT will need to download and install these libraries independent of SUNDIALS. SUNDIALS provides only the interfaces between itself and these libraries.

In the case of the direct methods CVDENSE and CVBAND the package includes an algorithm for the approximation of the Jacobian by difference quotients, but the user also has the option of supplying

the Jacobian (or an approximation to it) directly. When using the sparse direct linear solvers CVKLU and CVSUPERLUMT, the user must supply a routine for the Jacobian (or an approximation to it) in CSC format, since standard difference quotient approximations do not leverage the inherent sparsity of the problem. In the case of the Krylov iterative methods CVSPGMR, CVSPBCG, and CVSPTFQMR, the package includes an algorithm for the approximation by difference quotients of the product between the Jacobian matrix and a vector of appropriate length. Again, the user has the option of providing a routine for this operation. For the Krylov methods, the preconditioning must be supplied by the user, in two phases: setup (preprocessing of Jacobian data) 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 CVODE, offer considerable assistance in building preconditioners.

Each CVODE linear solver module consists of four routines, devoted to (1) memory allocation and initialization, (2) setup of the matrix data involved, (3) solution of the system, and (4) freeing of memory. The setup and solution phases are separate because the evaluation of Jacobians and preconditioners is done only periodically during the integration, and only as required to achieve convergence. The call list within the central CVODE module to each of the five associated functions is fixed, thus allowing the central module to be completely independent of the linear system method.

These modules are also decomposed in another way. With the exception of CVDIAG, each of the linear solver modules (CVDENSE etc.) consists of an interface built on top of a generic linear system solver (DENSE etc.). The interface deals with the use of the particular method in the CVODE context, whereas the generic solver is independent of the context. While some of the generic linear system solvers (DENSE, BAND, SPGMR, SPBCG, and SPTFQMR) were written with SUNDIALS in mind, they are intended to be usable anywhere as general-purpose solvers. This separation also allows for any generic solver to be replaced by an improved version, with no necessity to revise the CVODE package elsewhere.

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

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

Chapter 4

Using CVODE for C Applications

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

The sample programs described in the companion document [19] may also be helpful. Those codes may be used as templates (with the removal of some lines used in testing) and are included in the CVODE package.

Users with applications written in FORTRAN77 should see Chapter 5, which describes the FORTRAN/C interface module.

The user should be aware that not all linear solver modules are compatible with all NVECTOR implementations. For example, NVECTOR_PARALLEL is not compatible with the direct dense, direct band or direct sparse linear solvers since these linear solver modules need to form the complete system Jacobian. The following CVODE modules can only be used with either NVECTOR_SERIAL, NVECTOR_OPENMP or NVECTOR_PTHREADS: CVDENSE, CVBAND (using either the internal or the Lapack implementation), CVKLU, CVSUPERLUMT and CVBANDPRE. 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. Also, the preconditioner module CVBBDPRE can only be used with NVECTOR_PARALLEL.

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

4.1 Access to library and header files

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

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

- *libdir*/`libsundials_cvode.`*lib*,
- *libdir*/`libsundials_nvec*.`*lib* (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/cvode`
- *incdir*/`include/sundials`
- *incdir*/`include/nvector`

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

4.2 Data Types

The `sundials_types.h` file contains the definition of the type `realtype`, which is used by the SUNDIALS solvers for all floating-point data. The 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).

4.3 Header files

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

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

Note that `cvode.h` includes `sundials_types.h`, which defines the types `realtype` and `booleantype` and the constants `FALSE` and `TRUE`.

The calling program must also include an NVECTOR implementation header file, of the form `nvector_***.h`. See Chapter 6 for the appropriate name. This file in turn includes the header file `sundials_nvector.h` which defines the abstract `N_Vector` data type.

Finally, 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 solvers availble for use with CVODE are:

- `cvode_dense.h`, which is used with the dense direct linear solver;
- `cvode_band.h`, which is used with the band direct linear solver;
- `cvode_lapack.h`, which is used with Lapack implementations of dense or band direct linear solvers;
- `cvode_diag.h`, which is used with the diagonal linear solver;
- `cvode_klu.h`, which is used with the KLU sparse direct linear solver;

- `cvode_superlumt.h`, which is used with the SuperLU_MT threaded sparse direct linear solver;
- `cvode_spgmr.h`, which is used with the scaled, preconditioned GMRES Krylov linear solver SPGMR;
- `cvode_spbcgs.h`, which is used with the scaled, preconditioned Bi-CGStab Krylov linear solver SPBCG;
- `cvode_sptfqmr.h`, which is used with the scaled, preconditioned TFQMR Krylov solver SPTFQMR;

The header files for the dense and banded linear solvers (both internal and Lapack) include the file `cvode_direct.h`, which defines common functions. This in turn includes a file (`sundials_direct.h`) which defines the matrix type for these direct linear solvers (`DlsMat`), as well as various functions and macros acting on such matrices.

The header files for the KLU and SuperLU_MT sparse linear solvers include the file `cvode_sparse.h`, which defines common functions. This in turn includes a file (`sundials_sparse.h`) which defines the matrix type for these sparse direct linear solvers (`SlsMat`), as well as various functions and macros acting on such matrices.

The header files for the Krylov iterative solvers include `cvode_spils.h` which defines common functions and which in turn includes a header file (`sundials_iterative.h`) which enumerates the kind of preconditioning and (for the SPGMR solver only) the choices for the Gram-Schmidt process.

Other headers may be needed, according to the choice of preconditioner, etc. For example, in the `cvDiurnal_kry_p` example (see [19]), preconditioning is done with a block-diagonal matrix. For this, even though the CVSPGMR linear solver is used, the header `sundials_dense.h` is included for access to the underlying generic dense linear solver.

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 implementation used. For the steps that are not, refer to Chapter 6 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 `long int`.

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, use a call of the form `y0 = N_VMake_***(..., ydata)` if the `realtype` array `ydata` containing the initial values of y already exists. Otherwise, create a new vector by making a call of the form `y0 = N_VNew_***(...)`, and then set its elements by accessing the underlying data with a call of the form `ydata = N_VGetArrayPointer_***(y0)`. See §6.1-6.4 for details.

   For the *hypre* and PETSc vector wrappers, first create and initialize the underlying vector, and then create 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 §6.5 and §6.6 for details.

4. **Create CVODE object**

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

5. **Initialize CVODE solver**

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

6. **Specify integration tolerances**

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

7. **Set optional inputs**

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

8. **Attach linear solver module**

   If Newton iteration is chosen, initialize the linear solver module with one of the following calls (for details see §4.5.3):

   ```
   ier = CVDense(...);
   ier = CVBand(...);
   ier = CVLapackDense(...);
   ier = CVLapackBand(...);
   ier = CVKLU(...);
   ier = CVSuperLUMT(...);
   ier = CVDiag(...);
   ier = CVSpgmr(...);
   ier = CVSpbcg(...);
   ier = CVSptfqmr(...);
   ```

   NOTE: The direct (dense or band) and sparse linear solver options are usable only in a serial environment.

9. **Set linear solver optional inputs**

   Call `CV*Set*` functions from the selected linear solver module to change optional inputs specific to that linear solver. See §4.5.6 for details.

10. **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 §4.5.6.5 for relevant optional input calls.

11. **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.

12. **Get optional outputs**

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

13. **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);
    ```

14. **Free solver memory**

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

15. **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 in SUNDIALS packages and the vector implementations required for use. As an example, one cannot use the SUNDIALS package specific dense direct solver interfaces with the MPI-based vector implementation. However, as discussed in Chapter 10 the direct dense, direct band, and iterative spils solvers provided with SUNDIALS are written in a way that allows a user to develop their own solvers around them should a user so desire.

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

Linear Solver Interface	Serial	Parallel (MPI)	OpenMP	pThreads	*hypre* Vector	PETSc Vector	User Supplied
Dense	✓		✓	✓			✓
Band	✓		✓	✓			✓
LapackDense	✓		✓	✓			✓
LapackBand	✓		✓	✓			✓
KLU	✓		✓	✓			✓
SUPERLUMT	✓		✓	✓			✓
SPGMR	✓	✓	✓	✓	✓	✓	✓
SPFGMR	✓	✓	✓	✓	✓	✓	✓
SPBCG	✓	✓	✓	✓	✓	✓	✓
SPTFQMR	✓	✓	✓	✓	✓	✓	✓
User supplied	✓	✓	✓	✓	✓	✓	✓

4.5 User-callable functions

This section describes the CVODE functions that are called by the user to setup and then solve an IVP. Some of these are required. However, starting with §4.5.6, the functions listed involve optional inputs/outputs or restarting, and those paragraphs may be skipped for a casual use of CVODE. In any case, refer to §4.4 for the correct order of these calls.

On an error, each user-callable function returns a negative value and sends an error message to the error handler routine, which prints the message on `stderr` by default. However, the user can set a file as error output or can provide his own error handler function (see §4.5.6.1).

4.5.1 CVODE initialization and deallocation functions

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

`CVodeCreate`

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

Description The function `CVodeCreate` instantiates a CVODE 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 CVODE 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 CVODE.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODE memory block returned by `CVodeCreate`.
- `f` (`CVRhsFn`) is the C function which computes the right-hand side function f in the ODE. This function has the form `f(t, y, ydot, user_data)` (for full details see §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 CVODE memory block was not initialized through a previous call to `CVodeCreate`.
- `CV_MEM_FAIL` A memory allocation request has failed.
- `CV_ILL_INPUT` An input argument to `CVodeInit` has an illegal value.

Notes If an error occurred, `CVodeInit` also sends an error message to the error handler 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 CVODE memory block (of type `void *`).

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

4.5.2 CVODE tolerance specification functions

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

`CVodeSStolerances`

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

Description The function `CVodeSStolerances` specifies scalar relative and absolute tolerances.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODE memory block returned by `CVodeCreate`.
- `reltol` (`realtype`) is the scalar relative error tolerance.
- `abstol` (`realtype`) is the scalar absolute error tolerance.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeSStolerances` was successful.
- `CV_MEM_NULL` The CVODE memory block was not initialized through a previous call to `CVodeCreate`.
- `CV_NO_MALLOC` The allocation function `CVodeInit` has not been called.
- `CV_ILL_INPUT` One of the input tolerances was negative.

`CVodeSVtolerances`

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

Description The function `CVodeSVtolerances` specifies scalar relative tolerance and vector absolute tolerances.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODE memory block returned by `CVodeCreate`.
- `reltol` (`realtype`) is the scalar relative error tolerance.
- `abstol` (`N_Vector`) is the vector of absolute error tolerances.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeSVtolerances` was successful.
- `CV_MEM_NULL` The CVODE memory block was not initialized through a previous call to `CVodeCreate`.
- `CV_NO_MALLOC` The allocation function `CVodeInit` has not been called.
- `CV_ILL_INPUT` The relative error tolerance was negative or the absolute tolerance had a negative component.

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

`CVodeWFtolerances`

Call `flag = CVodeWFtolerances(cvode_mem, efun);`

Description The function `CVodeWFtolerances` specifies a user-supplied function `efun` that sets the multiplicative error weights W_i for use in the weighted RMS norm, which are normally defined by Eq. (2.6).

Arguments
- `cvode_mem` (`void *`) pointer to the CVODE memory block returned by `CVodeCreate`.
- `efun` (`CVEwtFn`) is the C function which defines the `ewt` vector (see §4.6.3).

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeWFtolerances` was successful.
- `CV_MEM_NULL` The CVODE memory block was not initialized through a previous call to `CVodeCreate`.
- `CV_NO_MALLOC` The allocation function `CVodeInit` has not been called.

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

(1) The scalar relative tolerance `reltol` is to be set to control relative errors. So `reltol` $= 10^{-4}$ means that errors are controlled to .01%. We do not recommend using `reltol` larger than 10^{-3}.

On the other hand, `reltol` should not be so small that it is comparable to the unit roundoff of the machine arithmetic (generally around 1.0E-15).

(2) The absolute tolerances `abstol` (whether scalar or vector) need to be set to control absolute errors when any components of the solution vector `y` may be so small that pure relative error control is meaningless. For example, if `y[i]` starts at some nonzero value, but in time decays to zero, then pure relative error control on `y[i]` makes no sense (and is overly costly) after `y[i]` is below some noise level. Then `abstol` (if scalar) or `abstol[i]` (if a vector) needs to be set to that noise level. If the different components have different noise levels, then `abstol` should be a vector. See the example `cvRoberts_dns` in the CVODE package, and the discussion of it in the CVODE Examples document [19]. 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 conservately, because they control the error committed on each individual time step. The final (global) errors are some sort of accumulation of those per-step errors. A good rule of thumb is to reduce the tolerances by a factor of .01 from the actual desired limits on errors. So if you want .01% accuracy (globally), a good choice is `reltol` $= 10^{-6}$. But in any case, it is a good idea to do a few experiments with the tolerances to see how the computed solution values vary as tolerances are reduced.

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

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

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

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

(4) Positivity and non-negativity constraints on components can be enforced by use of the recoverable error return feature in the user-supplied right-hand side function. However, because this option involves some extra overhead cost, it should only be exercised if the use of absolute tolerances to control the computed values is unsuccessful.

4.5.3 Linear solver specification functions

As previously explained, Newton iteration requires the solution of linear systems of the form (2.4). There are eight CVODE linear solvers currently available for this task: CVDENSE, CVBAND, CVKLU, CVSUPERLUMT, CVDIAG, CVSPGMR, CVSPBCG, and CVSPTFQMR.

The first two linear solvers are direct and derive their names from the type of approximation used for the Jacobian $J = \partial f/\partial y$; CVDENSE and CVBAND work with dense and banded approximations to J, respectively. The SUNDIALS suite includes both internal implementations of these two linear solvers and interfaces to Lapack implementations. Together, these linear solvers are referred to as CVDLS (from Direct Linear Solvers).

The second two linear solvers are sparse direct solvers based on Gaussian elimination, and require

user-supplied routines to construct the Jacobian $J = \partial f/\partial y$ in compressed-sparse-column format. The SUNDIALS suite does not include internal implementations of these solver libraries, instead requiring compilation of SUNDIALS to link with existing installations of these libraries (if either is missing, SUNDIALS will install without the corresponding interface routines). Together, these linear solvers are referred to as CVSLS (from Sparse Linear Solvers).

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

The last three CVODE linear solvers, CVSPGMR, CVSPBCG, and CVSPTFQMR, are Krylov iterative solvers, which use scaled preconditioned GMRES, scaled preconditioned Bi-CGStab, and scaled preconditioned TFQMR, respectively. Together, they are referred to as CVSPILS (from Scaled Preconditioned Iterative Linear Solvers).

With any of the Krylov methods, preconditioning can be done on the left only, on the right only, on both the left and the right, or not at all. 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_1P_2 approximates the Newton matrix $M = I - \gamma J$ of (2.5).

To specify a CVODE linear solver, after the call to `CVodeCreate` but before any calls to `CVode`, the user's program must call one of the functions `CVDense`/`CVLapackDense`, `CVBand`/`CVLapackBand`, `CVKLU`, `CVSuperLUMT`, `CVDiag`, `CVSpgmr`, `CVSpbcg`, or `CVSptfqmr`, as documented below. The first argument passed to these functions is the CVODE memory pointer returned by `CVodeCreate`. A call to one of these functions links the main CVODE integrator to a linear solver and allows the user to specify parameters which are specific to a particular solver, such as the half-bandwidths in the CVBAND case. The use of each of the 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 linear solver, as specified below.

In each case except the diagonal approximation case CVDIAG, the linear solver module used by CVODE is actually built on top of a generic linear system solver, which may be of interest in itself. These generic solvers, denoted DENSE, BAND, KLU, SUPERLUMT, SPGMR, SPBCG, and SPTFQMR, are described separately in Chapter 10.

`CVDense`

Call `flag = CVDense(cvode_mem, N);`

Description The function `CVDense` selects the CVDENSE linear solver and indicates the use of the internal direct dense linear algebra functions.

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

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`N` (`long int`) problem dimension.

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

`CVDLS_SUCCESS`	The CVDENSE initialization was successful.
`CVDLS_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CVDLS_ILL_INPUT`	The CVDENSE solver is not compatible with the current NVECTOR module.
`CVDLS_MEM_FAIL`	A memory allocation request failed.

Notes The CVDENSE linear solver is not compatible with all implementations of the NVECTOR module. Of the NVECTOR modules provided with SUNDIALS, only NVECTOR_SERIAL, NVECTOR_OPENMP and NVECTOR_PTHREADS are compatible.

`CVLapackDense`

Call `flag = CVLapackDense(cvode_mem, N);`

Description The function `CVLapackDense` selects the CVDENSE linear solver and indicates the use of Lapack functions.

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

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

`N` (`int`) problem dimension.

Return value The values of the returned `flag` (of type `int`) are identical to those of `CVDense`.

Notes Note that `N` is restricted to be of type `int` here, because of the corresponding type restriction in the Lapack solvers.

`CVBand`

Call `flag = CVBand(cvode_mem, N, mupper, mlower);`

Description The function `CVBand` selects the CVBAND linear solver and indicates the use of the internal direct band linear algebra functions.

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

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

`N` (`long int`) problem dimension.

`mupper` (`long int`) upper half-bandwidth of the problem Jacobian (or of the approximation of it).

`mlower` (`long int`) lower half-bandwidth of the problem Jacobian (or of the approximation of it).

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

`CVDLS_SUCCESS` The CVBAND initialization was successful.

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

`CVDLS_ILL_INPUT` The CVBAND solver is not compatible with the current NVECTOR module, or one of the Jacobian half-bandwidths is outside of its valid range ($0 \ldots$ `N`-1).

`CVDLS_MEM_FAIL` A memory allocation request failed.

Notes The CVBAND linear solver is not compatible with all implementations of the NVECTOR module. Of the NVECTOR modules provided with SUNDIALS, only NVECTOR_SERIAL, NVECTOR_OPENMP and NVECTOR_PTHREADS are compatible. The half-bandwidths are to be set such that the nonzero locations (i, j) in the banded (approximate) Jacobian satisfy $-$`mlower` $\leq j - i \leq$ `mupper`.

`CVLapackBand`

Call `flag = CVLapackBand(cvode_mem, N, mupper, mlower);`

Description The function `CVLapackBand` selects the CVBAND linear solver and indicates the use of Lapack functions.

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

Arguments The input arguments are identical to those of `CVBand`, except that `N`, `mupper`, and `mlower` are of type `int` here.

Return value The values of the returned `flag` (of type `int`) are identical to those of `CVBand`.

Notes Note that `N`, `mupper`, and `mlower` are restricted to be of type `int` here, because of the corresponding type restriction in the Lapack solvers.

CVKLU

Call `flag = CVKLU(cvode_mem, NP, NNZ, sparsetype);`

Description The function `CVKLU` selects the CVKLU linear solver and indicates the use of sparse-direct linear algebra functions.

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

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

`NP` (`int`) problem dimension.

`NNZ` (`int`) maximum number of nonzero entries in the system Jacobian.

`sparsetype` (`int`) sparse storage type of the system Jacobian. If `sparsetype` is set to `CSC_MAT` the solver will expect the Jacobian to be stored as a compressed sparse column matrix, and if `sparsetype=CSR_MAT` the solver will expect a compressed sparse row matrix. If neither option is chosen, the solver will exit with error.

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

`CVSLS_SUCCESS` The CVKLU initialization was successful.

`CVSLS_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CVSLS_ILL_INPUT` The CVKLU solver is not compatible with the current NVECTOR module.

`CVSLS_MEM_FAIL` A memory allocation request failed.

`CVSLS_PACKAGE_FAIL` A call to the KLU library returned a failure flag.

Notes The CVKLU linear solver is not compatible with all implementations of the NVECTOR module. Of the NVECTOR modules provided with SUNDIALS, only NVECTOR_SERIAL, NVECTOR_OPENMP and NVECTOR_PTHREADS are compatible.

CVSuperLUMT

Call `flag = CVSuperLUMT(cvode_mem, num_threads, N, NNZ);`

Description The function `CVSuperLUMT` selects the CVSUPERLUMT linear solver and indicates the use of sparse-direct linear algebra functions.

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

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

`num_threads` (`int`) the number of threads to use when factoring/solving the linear systems. Note that SuperLU_MT is thread-parallel only in the factorization routine.

`N` (`int`) problem dimension.

`NNZ` (`int`) maximum number of nonzero entries in the system Jacobian.

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

`CVSLS_SUCCESS` The CVSUPERLUMT initialization was successful.

`CVSLS_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CVSLS_ILL_INPUT` The CVSUPERLUMT solver is not compatible with the current NVECTOR module.

`CVSLS_MEM_FAIL` A memory allocation request failed.

`CVSLS_PACKAGE_FAIL` A call to the SuperLU_MT library returned a failure flag.

Notes The CVSUPERLUMT linear solver is not compatible with all implementations of the NVECTOR module. Of the NVECTOR modules provided with SUNDIALS, only NVECTOR_SERIAL, NVECTOR_OPENMP and NVECTOR_PTHREADS are compatible.

Performance will significantly degrade if the user applies the SuperLU_MT package

compiled with PThreads while using the NVECTOR_OPENMP module. If a user wants to use a threaded vector kernel with this thread-parallel solver, then SuperLU_MT should be compiled with openMP and the NVECTOR_OPENMP module should be used. Also, note that the expected benefit of using the threaded vector kernel is minimal compared to the potential benefit of the threaded solver, unless very long (greater than 100,000 entries) vectors are used.

`CVDiag`

Call `flag = CVDiag(cvode_mem);`

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

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

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

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

`CVDIAG_SUCCESS`	The CVDIAG initialization was successful.
`CVDIAG_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CVDIAG_ILL_INPUT`	The CVDIAG solver is not compatible with the current NVECTOR module.
`CVDIAG_MEM_FAIL`	A memory allocation request failed.

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

`CVSpgmr`

Call `flag = CVSpgmr(cvode_mem, pretype, maxl);`

Description The function `CVSpgmr` selects the CVSPGMR linear solver.

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

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

`pretype`	(`int`) specifies the preconditioning type and must be one of: `PREC_NONE`, `PREC_LEFT`, `PREC_RIGHT`, or `PREC_BOTH`.
`maxl`	(`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `CVSPILS_MAXL = 5`.

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

`CVSPILS_SUCCESS`	The CVSPGMR initialization was successful.
`CVSPILS_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CVSPILS_ILL_INPUT`	The preconditioner type `pretype` is not valid.
`CVSPILS_MEM_FAIL`	A memory allocation request failed.

Notes The CVSPGMR solver uses a scaled preconditioned GMRES iterative method to solve the linear system (2.4).

`CVSpbcg`

Call `flag = CVSpbcg(cvode_mem, pretype, maxl);`

Description The function `CVSpbcg` selects the CVSPBCG linear solver.

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

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

`pretype` (`int`) specifies the preconditioning type and must be one of: `PREC_NONE`, `PREC_LEFT`, `PREC_RIGHT`, or `PREC_BOTH`.

`maxl` (`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `CVSPILS_MAXL` = 5.

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

`CVSPILS_SUCCESS` The CVSPBCG initialization was successful.

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

`CVSPILS_ILL_INPUT` The preconditioner type `pretype` is not valid.

`CVSPILS_MEM_FAIL` A memory allocation request failed.

Notes The CVSPBCG solver uses a scaled preconditioned Bi-CGStab iterative method to solve the linear system (2.4).

`CVSptfqmr`

Call `flag = CVSptfqmr(cvode_mem, pretype, maxl);`

Description The function `CVSptfqmr` selects the CVSPTFQMR linear solver.

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

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

`pretype` (`int`) specifies the preconditioning type and must be one of: `PREC_NONE`, `PREC_LEFT`, `PREC_RIGHT`, or `PREC_BOTH`.

`maxl` (`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `CVSPILS_MAXL` = 5.

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

`CVSPILS_SUCCESS` The CVSPTFQMR initialization was successful.

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

`CVSPILS_ILL_INPUT` The preconditioner type `pretype` is not valid.

`CVSPILS_MEM_FAIL` A memory allocation request failed.

Notes The CVSPTFQMR solver uses a scaled preconditioned TFQMR iterative method to solve the linear system (2.4).

4.5.4 Rootfinding initialization function

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

`CVodeRootInit`

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

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

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

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

`g` (`CVRootFn`) is the C function which defines the `nrtfn` functions $g_i(t, y)$ whose roots are sought. See §4.6.4 for details.

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

`CV_SUCCESS` The call to `CVodeRootInit` was successful.

`CV_MEM_NULL` The `cvode_mem` argument was `NULL`.

CV_MEM_FAIL A memory allocation failed.

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

Notes If a new IVP is to be solved with a call to CVodeReInit, where the new IVP has no rootfinding problem but the prior one did, then call CVodeRootInit with nrtfn= 0.

4.5.5 CVODE solver function

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

CVode

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

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

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

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

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

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

- CV_SUCCESS CVode succeeded and no roots were found.
- CV_TSTOP_RETURN CVode succeeded by reaching the stopping point specified through the optional input function CVodeSetStopTime (see §4.5.6.1).
- 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 CVODE memory was not allocated by a call to CVodeInit.
- CV_ILL_INPUT One of the inputs to CVode was illegal, or some other input to the solver was either illegal or missing. The latter category includes the following situations: (a) The tolerances have not been set. (b) A component of the error weight vector became zero during internal time-stepping. (c) The linear solver initialization function (called by the user after calling CVodeCreate) failed to set the linear solver-specific lsolve field in cvode_mem. (d) A root of one of the root functions was found both at a point t and also very near t. In any case, the user should see the error message for details.
- CV_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_RHSFUNC_FAIL	The right-hand side function failed in an unrecoverable manner.
CV_FIRST_RHSFUNC_FAIL	The right-hand side function had a recoverable error at the first call.
CV_REPTD_RHSFUNC_ERR	Convergence test failures occurred too many times due to repeated recoverable errors in the right-hand side function. This flag will also be returned if the right-hand side function had repeated recoverable errors during the estimation of an initial step size.
CV_UNREC_RHSFUNC_ERR	The right-hand function had a recoverable error, but no recovery was possible. This failure mode is rare, as it can occur only if the right-hand side function fails recoverably after an error test failed while at order one.
CV_RTFUNC_FAIL	The rootfinding function failed.

Notes

The vector yout can occupy the same space as the vector y0 of initial conditions that was passed to CVodeInit.

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

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

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

4.5.6 Optional input functions

There are numerous optional input parameters that control the behavior of the CVODE solver. CVODE provides functions that can be used to change these optional input parameters from their default values. Table 4.2 lists all optional input functions in CVODE which are then described in detail in the remainder of this section, begining with those for the main CVODE solver and continuing with those for the linear solver modules. Note that the diagonal linear solver module has no optional inputs. For the most casual use of CVODE, the reader can skip to §4.6.

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

4.5.6.1 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.

Table 4.2: Optional inputs for CVODE, CVDLS, CVSLS, and CVSPILS

Optional input	Function name	Default
CVODE main solver		
Pointer to an error file	`CVodeSetErrFile`	`stderr`
Error handler function	`CVodeSetErrHandlerFn`	internal fn.
User data	`CVodeSetUserData`	`NULL`
Maximum order for BDF method	`CVodeSetMaxOrd`	5
Maximum order for Adams method	`CVodeSetMaxOrd`	12
Maximum no. of internal steps before t_{out}	`CVodeSetMaxNumSteps`	500
Maximum no. of warnings for $t_n + h = t_n$	`CVodeSetMaxHnilWarns`	10
Flag to activate stability limit detection	`CVodeSetStabLimDet`	`FALSE`
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
Direction of zero-crossing	`CVodeSetRootDirection`	both
Disable rootfinding warnings	`CVodeSetNoInactiveRootWarn`	none
CVDLS linear solvers		
Dense Jacobian function	`CVDlsSetDenseJacFn`	DQ
Band Jacobian function	`CVDlsSetBandJacFn`	DQ
CVSLS linear solvers		
Sparse Jacobian function	`CVSlsSetSparseJacFn`	none
Sparse matrix ordering algorithm	`CVKLUSetOrdering`	1 for `COLAMD`
Sparse matrix ordering algorithm	`CVSuperLUMTSetOrdering`	3 for `COLAMD`
CVSPILS linear solvers		
Preconditioner functions	`CVSpilsSetPreconditioner`	NULL, NULL
Jacobian-times-vector function	`CVSpilsSetJacTimesVecFn`	DQ
Preconditioning type	`CVSpilsSetPrecType`	none
Ratio between linear and nonlinear tolerances	`CVSpilsSetEpsLin`	0.05
Type of Gram-Schmidt orthogonalization[(a)]	`CVSpilsSetGSType`	modified GS
Maximum Krylov subspace size[(b)]	`CVSpilsSetMaxl`	5

[(a)] Only for CVSPGMR
[(b)] Only for CVSPBCG and CVSPTFQMR

`CVodeSetErrFile`

Call `flag = CVodeSetErrFile(cvode_mem, errfp);`

Description The function `CVodeSetErrFile` specifies a pointer to the file where all CVODE messages should be directed when the default CVODE error handler function is used.

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

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

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

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

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

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

`CVodeSetErrHandlerFn`

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

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

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

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

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

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

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

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

`CVodeSetMaxOrd`

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

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

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

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

Notes The default value is `ADAMS_Q_MAX = 12` for the Adams-Moulton method and `BDF_Q_MAX = 5` for the BDF method. Since `maxord` affects the memory requirements for the internal CVODE memory block, its value cannot be increased past its previous value.

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

`CVodeSetMaxNumSteps`

Call `flag = CVodeSetMaxNumSteps(cvode_mem, mxsteps);`

Description The function `CVodeSetMaxNumSteps` specifies the maximum number of steps to be taken by the solver in its attempt to reach the next output time.

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`mxsteps` (`long int`) maximum allowed number of steps.

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

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

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

`CVodeSetMaxHnilWarns`

Call `flag = CVodeSetMaxHnilWarns(cvode_mem, mxhnil);`

Description The function `CVodeSetMaxHnilWarns` specifies the maximum number of messages issued by the solver warning that $t + h = t$ on the next internal step.

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`mxhnil` (`int`) maximum number of warning messages (> 0).

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

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

`CVodeSetStabLimDet`

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

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

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

`stldet` (`booleantype`) flag controlling stability limit detection (`TRUE` = on; `FALSE` = 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 `FALSE`. If `stldet = TRUE` 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 CVODE memory block.
`hin` (`realtype`) value of the initial step size to be attempted. Pass 0.0 to use the default value.

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

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

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

`CVodeSetMinStep`

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

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

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

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

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

Notes The default value is 0.0.

`CVodeSetMaxStep`

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

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

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

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

`CV_SUCCESS` The optional value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`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 CVODE memory block.
`tstop` (`realtype`) value of the independent variable past which the solution should not proceed.

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

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

`CVodeSetMaxErrTestFails`

Call `flag = CVodeSetMaxErrTestFails(cvode_mem, maxnef);`

Description The function `CVodeSetMaxErrTestFails` specifies the maximum number of error test failures permitted in attempting one step.

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`maxnef` (`int`) maximum number of error test failures allowed on one step (> 0).

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

Notes The default value is 7.

`CVodeSetMaxNonlinIters`

Call `flag = CVodeSetMaxNonlinIters(cvode_mem, maxcor);`

Description The function `CVodeSetMaxNonlinIters` specifies the maximum number of nonlinear solver iterations permitted per step.

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`maxcor` (`int`) maximum number of nonlinear solver iterations allowed per step (> 0).

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

Notes The default value is 3.

`CVodeSetMaxConvFails`

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

Description The function `CVodeSetMaxConvFails` specifies the maximum number of nonlinear solver convergence failures permitted during one step.

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`maxncf` (`int`) maximum number of allowable nonlinear solver convergence failures per step (> 0).

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

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes The default value is 10.

CVodeSetNonlinConvCoef

Call `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 CVODE memory block.

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

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

Notes The default value is 0.1.

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 CVODE 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.

4.5.6.2 Dense/band direct linear solvers optional input functions

The CVDENSE solver needs a function to compute a dense approximation to the Jacobian matrix $J(t, y)$. This function must be of type CVDlsDenseJacFn. The user can supply his/her own dense Jacobian function, or use the default internal difference quotient approximation that comes with the CVDENSE solver. To specify a user-supplied Jacobian function `djac`, CVDENSE provides the function CVDlsSetDenseJacFn. The CVDENSE solver passes the pointer `user_data` to the dense 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.

CVDlsSetDenseJacFn

Call `flag = CVDlsSetDenseJacFn(cvode_mem, djac);`

Description The function CVDlsSetDenseJacFn specifies the dense Jacobian approximation function to be used.

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

`djac` (CVDlsDenseJacFn) user-defined dense 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 CVDENSE linear solver has not been initialized.

Notes By default, CVDENSE uses an internal difference quotient function. If NULL is passed to djac, this default function is used.

The function type CVDlsDenseJacFn is described in §4.6.5.

The CVBAND solver needs a function to compute a banded approximation to the Jacobian matrix $J(t, y)$. This function must be of type CVDlsBandJacFn. The user can supply his/her own banded Jacobian approximation function, or use the default internal difference quotient approximation that comes with the CVBAND solver. To specify a user-supplied Jacobian function bjac, CVBAND provides the function CVDlsSetBandJacFn. The CVBAND solver passes the pointer user_data to the banded Jacobian approximation 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.

CVDlsSetBandJacFn

Call `flag = CVDlsSetBandJacFn(cvode_mem, bjac);`

Description The function CVDlsSetBandJacFn specifies the banded Jacobian approximation function to be used.

Arguments
cvode_mem	(void *) pointer to the CVODE memory block.
bjac	(CVBandJacFn) user-defined banded 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 CVBAND linear solver has not been initialized.

Notes By default, CVBAND uses an internal difference quotient function. If NULL is passed to bjac, this default function is used.

The function type CVBandJacFn is described in §4.6.6.

4.5.6.3 Sparse direct linear solvers optional input functions

The CVKLU and CVSUPERLUMT solvers require a function to compute a compressed-sparse-column approximation to the Jacobian matrix $J(t, y)$. This function must be of type CVSlsSparseJacFn. The user must supply a custom sparse Jacobian function since a difference-quotient approximation would not leverage the underlying sparse matrix structure of the problem. To specify a user-supplied Jacobian function sjac, CVKLU and CVSUPERLUMT provide the function CVSlsSetSparseJacFn. The CVKLU and CVSUPERLUMT solvers pass the pointer user_data to the sparse Jacobian function. This mechanism 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.

CVSlsSetSparseJacFn

Call `flag = CVSlsSetSparseJacFn(cvode_mem, sjac);`

Description The function CVSlsSetSparseJacFn specifies the sparse Jacobian approximation function to be used.

Arguments
cvode_mem	(void *) pointer to the CVODE memory block.
sjac	(CVSlsSparseJacFn) user-defined sparse Jacobian approximation function.

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

CVSLS_SUCCESS The sparse Jacobian routine pointer has been successfully set.

CVSLS_MEM_NULL The cvode_mem pointer is NULL.

CVSLS_LMEM_NULL The CVKLU or CVSUPERLUMT linear solver has not been initialized.

Notes The function type CVSlsSparseJacFn is described in §4.6.7.

When using a sparse direct solver, there may be instances when the number of state variables does not change, but the number of nonzeroes in the Jacobian does change. In this case, for the CVKLU solver, we provide the following reinitialization function. This function reinitializes the Jacobian matrix memory for the new number of nonzeroes and sets 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 where the structure of the linear system has changed, requiring a new symbolic (and numeric) factorization.

CVKLUReInit

Call `flag = CVKLUReInit(cv_mem, n, nnz, reinit_type);`

Description The function CVKLUReInit reinitializes Jacobian matrix memory and flags for new symbolic and numeric KLU factorizations.

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

`n` (`int`) number of state variables in the system.

`nnz` (`int`) number of nonzeroes in the Jacobian matrix.

`reinit_type` (`int`) type of reinitialization:

1 The Jacobian matrix will be destroyed and a new one will be allocated based on the `nnz` value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup.

2 Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of `nnz` given in the prior call to CVKLU.

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

CVSLS_SUCCESS The reinitialization succeeded.

CVSLS_MEM_NULL The cv_mem pointer is NULL.

CVSLS_LMEM_NULL The CVKLU linear solver has not been initialized.

CVSLS_ILL_INPUT The given reinit_type has an illegal value.

CVSLS_MEM_FAIL A memory allocation failed.

Notes The default value for reinit_type is 2.

Both the CVKLU and CVSUPERLUMT solvers can apply reordering algorithms to minimize fill-in for the resulting sparse *LU* decomposition internal to the solver. The approximate minimal degree ordering for nonsymmetric matrices given by the COLAMD algorithm is the default algorithm used within both solvers, but alternate orderings may be chosen through one of the following two functions. The input values to these functinos are the numeric values used in the respective packages, and the user-supplied value will be passed directly to the package.

CVKLUSetOrdering

Call `flag = CVKLUSetOrdering(cv_mem, ordering_choice);`

Description The function CVKLUSetOrdering specifies the ordering algorithm used by CVKLU for reducing fill.

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

`ordering_choice` (`int`) flag denoting algorithm choice:

0 AMD

1 COLAMD
2 natural ordering

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

`CVSLS_SUCCESS`	The optional value has been successfully set.
`CVSLS_MEM_NULL`	The `cv_mem` pointer is `NULL`.
`CVSLS_ILL_INPUT`	The supplied value of `ordering_choice` is illegal.

Notes The default ordering choice is 1 for `COLAMD`.

`CVSuperLUMTSetOrdering`

Call `flag = CVSuperLUMTSetOrdering(cv_mem, ordering_choice);`

Description The function `CVSuperLUMTSetOrdering` specifies the ordering algorithm used by CVSUPERLUMT for reducing fill.

Arguments `cv_mem` (`void *`) pointer to the CVODE memory block.
`ordering_choice` (`int`) flag denoting algorithm choice:

0 natural ordering
1 minimal degree ordering on $J^T J$
2 minimal degree ordering on $J^T + J$
3 COLAMD

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

`CVSLS_SUCCESS`	The optional value has been successfully set.
`CVSLS_MEM_NULL`	The `cv_mem` pointer is `NULL`.
`CVSLS_ILL_INPUT`	The supplied value of `ordering_choice` is illegal.

Notes The default ordering choice is 3 for `COLAMD`.

4.5.6.4 Iterative linear solvers optional input functions

If any preconditioning is to be done within one of the CVSPILS linear solvers, 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.

Ther CVSPILS solvers require a function to compute an approximation to the product between the Jacobian matrix $J(t, y)$ and a vector v. The user can supply his/her own Jacobian-times-vector approximation function, or use the default internal difference quotient function that comes with the CVSPILS solvers. A user-defined Jacobian-vector function must be of type `CVSpilsJacTimesVecFn` and can be specified through a call to `CVSpilsSetJacTimesVecFn` (see §4.6.8 for specification details). As with the preconditioner user-supplied 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 function `jtimes` each time it is called.

`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 CVODE memory block.
- `psetup` (`CVSpilsPrecSetupFn`) user-defined preconditioner setup function. Pass `NULL` if no setup is to be done.
- `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.

Notes The function type `CVSpilsPrecSolveFn` is described in §4.6.9. The function type `CVSpilsPrecSetupFn` is described in §4.6.10.

`CVSpilsSetJacTimesVecFn`

Call `flag = CVSpilsSetJacTimesVecFn(cvode_mem, jtimes);`

Description The function `CVSpilsSetJacTimesFn` specifies the Jacobian-vector function to be used.

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

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 `CVSpilsJacTimesVecFn` is described in §4.6.8.

`CVSpilsSetPrecType`

Call `flag = CVSpilsSetPrecType(cvode_mem, pretype);`

Description The function `CVSpilsSetPrecType` resets the type of preconditioning to be used.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODE memory block.
- `pretype` (`int`) specifies the type of preconditioning and must be one of: `PREC_NONE`, `PREC_LEFT`, `PREC_RIGHT`, or `PREC_BOTH`.

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 preconditioner type `pretype` is not valid.

Notes The preconditioning type is initially set in the call to the linear solver's specification function (see §4.5.3). This function call is needed only if `pretype` is being changed from its original value.

`CVSpilsSetGSType`

Call `flag = CVSpilsSetGSType(cvode_mem, gstype);`

Description The function `CVSpilsSetGSType` specifies the Gram-Schmidt orthogonalization to be used with the CVSPGMR solver (one of the enumeration constants `MODIFIED_GS` or `CLASSICAL_GS`). These correspond to using modified Gram-Schmidt and classical Gram-Schmidt, respectively.

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`gstype` (`int`) type of Gram-Schmidt orthogonalization.

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 value of `gstype` is not valid.

Notes The default value is `MODIFIED_GS`.

This option is available only for the CVSPGMR linear solver.

`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 CVODE 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.

Passing a value `eplifac`$= 0.0$ also indicates using the default value.

`CVSpilsSetMaxl`

Call `flag = CVSpilsSetMaxl(cv_mem, maxl);`

Description The function `CVSpilsSetMaxl` resets the maximum Krylov subspace dimension for the Bi-CGStab or TFQMR methods.

Arguments `cv_mem` (`void *`) pointer to the CVODE memory block.
`maxl` (`int`) maximum dimension of the Krylov subspace.

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 current linear solver is SPGMR.

Notes The maximum subspace dimension is initially specified in the call to the linear solver specification function (see §4.5.3). This function call is needed only if `maxl` is being changed from its previous value.

An input value `maxl` ≤ 0 will result in the default value, 5.

This option is available only for the CVSPBCG and CVSPTFQMR linear solvers.

4.5.6.5 Rootfinding optional input functions

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

`CVodeSetRootDirection`

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

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

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

`rootdir` (`int *`) state array of length `nrtfn`, the number of root functions g_i, as 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 CVODE memory block.

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

`CV_SUCCESS` The optional value has been successfully set.

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

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

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 CVODE.

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

`CVodeGetDky`

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

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

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

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

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

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

4.5.8 Optional output functions

CVODE provides an extensive set of functions that can be used to obtain solver performance information. Table 4.3 lists all optional output functions in CVODE, which are then described in detail in the remainder of this section.

Some of the optional outputs, especially the various counters, can be very useful in determining how successful the CVODE solver is in doing its job. For example, the counters `nsteps` and `nfevals` provide a rough measure of the overall cost of a given run, and can be compared among runs with differing input options to suggest which set of options is most efficient. The ratio `nniters/nsteps` measures the performance of the 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 Main solver optional output functions

CVODE provides several user-callable functions that can be used to obtain different quantities that may be of interest to the user, such as solver workspace requirements, solver performance statistics, as well as additional data from the CVODE memory block (a suggested tolerance scaling factor, the error weight vector, and the vector of estimated local errors). Functions are also provided to extract statistics related to the performance of the CVODE nonlinear solver used. As a convenience, additional information extraction functions provide the optional outputs in groups. These optional output functions are described next.

`CVodeGetWorkSpace`

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

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

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

Table 4.3: Optional outputs from CVODE, CVDLS, CVDIAG, CVSLS, and CVSPILS

Optional output	Function name
CVODE main solver	
Size of CVODE real and integer workspaces	CVodeGetWorkSpace
Cumulative number of internal steps	CVodeGetNumSteps
No. of calls to r.h.s. function	CVodeGetNumRhsEvals
No. of calls to linear solver setup function	CVodeGetNumLinSolvSetups
No. of local error test failures that have occurred	CVodeGetNumErrTestFails
Order used during the last step	CVodeGetLastOrder
Order to be attempted on the next step	CVodeGetCurrentOrder
No. of order reductions due to stability limit detection	CVodeGetNumStabLimOrderReds
Actual initial step size used	CVodeGetActualInitStep
Step size used for the last step	CVodeGetLastStep
Step size to be attempted on the next step	CVodeGetCurrentStep
Current internal time reached by the solver	CVodeGetCurrentTime
Suggested factor for tolerance scaling	CVodeGetTolScaleFactor
Error weight vector for state variables	CVodeGetErrWeights
Estimated local error vector	CVodeGetEstLocalErrors
No. of nonlinear solver iterations	CVodeGetNumNonlinSolvIters
No. of nonlinear convergence failures	CVodeGetNumNonlinSolvConvFails
All CVODE integrator statistics	CVodeGetIntegratorStats
CVODE nonlinear solver statistics	CVodeGetNonlinSolvStats
Array showing roots found	CvodeGetRootInfo
No. of calls to user root function	CVodeGetNumGEvals
Name of constant associated with a return flag	CVodeGetReturnFlagName
CVDLS linear solvers	
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	
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
CVSLS linear solvers	
No. of Jacobian evaluations	CVSlsGetNumJacEvals
Last return from a linear solver function	CVSlsGetLastFlag
Name of constant associated with a return flag	CVSlsGetReturnFlagName
CVSPILS linear solvers	
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 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

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` $+N_r$;

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` $+N_i$;

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, 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` $= 50$

`CVodeGetNumSteps`

Call `flag = CVodeGetNumSteps(cvode_mem, &nsteps);`

Description The function `CVodeGetNumSteps` returns the cumulative number of internal steps taken by the solver (total so far).

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

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

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

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

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

`CVodeGetNumRhsEvals`

Call `flag = CVodeGetNumRhsEvals(cvode_mem, &nfevals);`

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

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

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

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

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

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

Notes The `nfevals` value returned by `CVodeGetNumRhsEvals` does not account for calls made to `f` by a linear solver or preconditioner module.

`CVodeGetNumLinSolvSetups`

Call `flag = CVodeGetNumLinSolvSetups(cvode_mem, &nlinsetups);`

Description The function `CVodeGetNumLinSolvSetups` returns the number of calls made to the linear solver's setup function.

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`nlinsetups` (`long int`) number of calls made to the linear solver setup function.

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

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

`CVodeGetNumErrTestFails`

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

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

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

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

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

`CVodeGetLastOrder`

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

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

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

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

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

`CVodeGetCurrentOrder`

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

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

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

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

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

`CVodeGetLastStep`

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

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

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

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

`CVodeGetCurrentStep`

Call `flag = CVodeGetCurrentStep(cvode_mem, &hcur);`

Description The function `CVodeGetCurrentStep` returns the integration step size to be attempted on the next internal step.

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`hcur` (`realtype`) step size to be attempted on the next internal step.

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

`CVodeGetActualInitStep`

Call `flag = CVodeGetActualInitStep(cvode_mem, &hinused);`

Description The function `CVodeGetActualInitStep` returns the value of the integration step size used on the first step.

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`hinused` (`realtype`) actual value of initial step size.

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

Notes Even if the value of the initial integration step size was specified by the user through a call to `CVodeSetInitStep`, this value might have been changed by CVODE to ensure that the step size is within the prescribed bounds ($h_{\min} \le h_0 \le h_{\max}$), 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 CVODE memory block.
`tcur` (`realtype`) current internal time reached.

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

`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 CVODE memory block.
`nslred` (`long int`) number of order reductions due to stability limit detection.

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

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

`CVodeGetTolScaleFactor`

Call `flag = CVodeGetTolScaleFactor(cvode_mem, &tolsfac);`

Description The function `CVodeGetTolScaleFactor` returns a suggested factor by which the user's tolerances should be scaled when too much accuracy has been requested for some internal step.

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`tolsfac` (`realtype`) suggested scaling factor for user-supplied tolerances.

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

`CVodeGetErrWeights`

Call `flag = CVodeGetErrWeights(cvode_mem, eweight);`

Description The function `CVodeGetErrWeights` returns the solution error weights at the current time. These are the reciprocals of the W_i given by (2.6).

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`eweight` (`N_Vector`) solution error weights at the current time.

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

Notes The user must allocate memory for `eweight`.

`CVodeGetEstLocalErrors`

Call `flag = CVodeGetEstLocalErrors(cvode_mem, ele);`

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

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`ele` (`N_Vector`) estimated local errors.

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

Notes The user must allocate memory for `ele`.

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

The `ele` vector, 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 CVODE integrator statistics as a group.

Arguments

`cvode_mem`	(`void *`) pointer to the CVODE memory block.
`nsteps`	(`long int`) number of steps taken by CVODE.
`nfevals`	(`long int`) number of calls to the user's `f` function.
`nlinsetups`	(`long int`) number of calls made to the linear solver setup function.
`netfails`	(`long int`) number of error test failures.
`qlast`	(`int`) method order used on the last internal step.
`qcur`	(`int`) method order to be used on the next internal step.
`hinused`	(`realtype`) actual value of initial step size.
`hlast`	(`realtype`) step size taken on the last internal step.
`hcur`	(`realtype`) step size to be attempted on the next internal step.
`tcur`	(`realtype`) current internal time reached.

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

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

`CVodeGetNumNonlinSolvIters`

Call
```
flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nniters);
```

Description The function `CVodeGetNumNonlinSolvIters` returns the number of nonlinear (functional or Newton) iterations performed.

Arguments

`cvode_mem`	(`void *`) pointer to the CVODE memory block.
`nniters`	(`long int`) number of nonlinear iterations performed.

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

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

`CVodeGetNumNonlinSolvConvFails`

Call
```
flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &nncfails);
```

Description The function `CVodeGetNumNonlinSolvConvFails` returns the number of nonlinear convergence failures that have occurred.

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

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

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

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

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

`CVodeGetNonlinSolvStats`

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

Description The function `CVodeGetNonlinSolvStats` returns the CVODE nonlinear solver statistics as a group.

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

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

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

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

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

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

`CVodeGetReturnFlagName`

Call `name = CVodeGetReturnFlagName(flag);`

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

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

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

4.5.8.2 Rootfinding optional output functions

There are two optional output functions associated with rootfinding.

`CVodeGetRootInfo`

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

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

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

`rootsfound` (`int *`) array of length `nrtfn` with the indices of the user functions g_i found to have a root. For $i = 0, \ldots,$`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 CVODE memory block.
`ngevals` (`long int`) number of calls made to the user's function `g` thus far.

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

4.5.8.3 Dense/band direct linear solvers 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 here (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 a CVDLS linear solver (CVDENSE or CVBAND).

Arguments `cvode_mem` (`void *`) pointer to the CVODE 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 For the CVDENSE linear solver, in terms of the problem size N, the actual size of the real workspace is $2N^2$ `realtype` words, and the actual size of the integer workspace is N integer words. For the CVBAND linear solver, in terms of N and Jacobian half-bandwidths, the actual size of the real workspace is (2 `mupper`+3 `mlower`+2) N `realtype` words, and the actual size of the integer workspace is N integer words.

`CVDlsGetNumJacEvals`

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

Description The function `CVDlsGetNumJacEvals` returns the number of calls made to the CVDLS (dense or band) Jacobian approximation function.

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`njevals` (`long int`) the number of calls to the Jacobian function.

Return value The return value `flag` (of type `int`) is one of
`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 (dense or band) Jacobian approximation.

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

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

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

`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 the default internal difference quotient function 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 CVODE 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 CVDENSE 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. For all other failures, `lsflag` is negative.

`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".

4.5.8.4 Diagonal linear solver optional output functions

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

`CVDiagGetWorkSpace`

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

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

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

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

`CVDIAG_SUCCESS` The optional output 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 CVODE memory block.
`nfevalsLS` (`long int`) the number of calls made to the user-supplied right-hand side function.

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

`CVDIAG_SUCCESS` The optional output value has been successfully set.
`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 CVODE 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.

4.5.8.5 Sparse direct linear solvers optional output functions

The following optional outputs are available from the CVSLS modules: number of calls to the Jacobian routine and last return value from a CVSLS function.

`CVSlsGetNumJacEvals`

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

Description The function `CVSlsGetNumJacEvals` returns the number of calls made to the CVSLS sparse Jacobian approximation function.

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`njevals` (`long int`) the number of calls to the Jacobian function.

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

`CVSLS_SUCCESS`	The optional output value has been successfully set.
`CVSLS_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CVSLS_LMEM_NULL`	The CVSLS linear solver has not been initialized.

`CVSlsGetLastFlag`

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

Description The function `CVSlsGetLastFlag` returns the last return value from a CVSLS routine.

Arguments `cvode_mem` (`void *`) pointer to the CVODE memory block.
`lsflag` (`long int`) the value of the last return flag from a CVSLS function.

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

`CVSLS_SUCCESS`	The optional output value has been successfully set.
`CVSLS_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CVSLS_LMEM_NULL`	The CVSLS linear solver has not been initialized.

Notes

`CVSlsGetReturnFlagName`

Call `name = CVSlsGetReturnFlagName(lsflag);`

Description The function `CVSlsGetReturnFlagName` returns the name of the CVSLS constant corresponding to `lsflag`.

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

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

4.5.8.6 Iterative linear solvers 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 product routine, 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 here (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 CVODE 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 In terms of the problem size N and maximum subspace size `maxl`, the actual size of the real workspace is roughly:
(`maxl`+5) $*$ N+ `maxl` $*$(`maxl`+4) + 1 `realtype` words for CVSPGMR,
$9 * N$ `realtype` words for CVSPBCG,
and $11 * N$ `realtype` words for CVSPTFQMR.

In a parallel setting, the above values are global, 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 CVODE 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 CVODE memory block.
- `nlcfails` (`long int`) the current number of linear convergence failures.

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

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 = FALSE.

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

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

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

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

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 made to the Jacobian-vector function, jtimes.

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

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

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 CVODE 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 CVODE 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 SPGMR_PSET_FAIL_UNREC, SPBCG_PSET_FAIL_UNREC, or SPTFQMR_PSET_FAIL_UNREC.

If the CVSPGMR solve function failed (CVode returned CV_LSOLVE_FAIL), lsflag contains the error return flag from SpgmrSolve and will be one of: SPGMR_MEM_NULL, indicating that the SPGMR memory is NULL; SPGMR_ATIMES_FAIL_UNREC, indicating an unrecoverable failure in the $J*v$ function; SPGMR_PSOLVE_FAIL_UNREC, indicating that the preconditioner solve function psolve failed unrecoverably; SPGMR_GS_FAIL, indicating a failure in the Gram-Schmidt procedure; or SPGMR_QRSOL_FAIL, indicating that the matrix R was found to be singular during the QR solve phase.

If the CVSPBCG solve function failed (CVode returned CV_LSOLVE_FAIL), lsflag contains the error return flag from SpbcgSolve and will be one of: SPBCG_MEM_NULL, indicating that the SPBCG memory is NULL; SPBCG_ATIMES_FAIL_UNREC, indicating an unrecoverable failure in the $J*v$ function; or SPBCG_PSOLVE_FAIL_UNREC, indicating that the preconditioner solve function psolve failed unrecoverably.

If the CVSPTFQMR solve function failed (CVode returned CV_LSOLVE_FAIL), lsflag contains the error return flag from SptfqmrSolve and will be one of: SPTFQMR_MEM_NULL, indicating that the SPTFQMR memory is NULL; SPTFQMR_ATIMES_FAIL_UNREC, indicating an unrecoverable failure in the $J*v$ function; or SPTFQMR_PSOLVE_FAIL_UNREC, indicating that the preconditioner solve function psolve failed unrecoverably.

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.

4.5.9 CVODE reinitialization function

The function CVodeReInit reinitializes the main CVODE solver for the solution of a new problem, where a prior call to CVodeInit been made. The new problem must have the same size as the previous one. CVodeReInit performs the same input checking and initializations that CVodeInit does, but does no

memory allocation, as it assumes that the existing internal memory is sufficient for the new problem. A call to CVodeReInit deletes the solution history that was stored internally during the previous integration. Following a successful call to CVodeReInit, call CVode again for the solution of the new problem.

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

If there are changes to the linear solver specifications, make the appropriate CV*** calls, as described in §4.5.3. Otherwise, all solver inputs set previously remain in effect.

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

CVodeReInit

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

Description The function CVodeReInit provides required problem specifications and reinitializes CVODE.

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

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

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

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) a function that provides 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 CVODE 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 CVODE.

A recoverable failure error return from the `CVRhsFn` is typically used to flag a value of the dependent variable `y` that is "illegal" in some way (e.g., negative where only a non-negative value is physically meaningful). If such a return is made, CVODE 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.)

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

4.6.2 Error message handler function

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

CVErrHandlerFn

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

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

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

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

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

4.6.3 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. (2.6). The function type `CVEwtFn` is defined as follows:

`CVEwtFn`

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

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

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

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

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

The error weight vector must have all components positive. It is the user's 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 CVODE.

4.6.5 Jacobian information (direct method with dense Jacobian)

If the direct linear solver with dense treatment of the Jacobian is used (i.e., `CVDense` or `CVLapackDense` is called in Step 8 of §4.4), the user may provide a function of type `CVDlsDenseJacFn` defined by:

`CVDlsDenseJacFn`

Definition
```
typedef (*CVDlsDenseJacFn)(long int N, realtype t, N_Vector y,
                           N_Vector fy, DlsMat Jac, void *user_data,
                           N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
```

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

Arguments

`N`	is the problem size.
`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 dense Jacobian matrix (of type `DlsMat`).
`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 `CVDlsDenseJacFn` as temporary storage or work space.

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

Notes 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`). Only nonzero elements need to be loaded into `Jac` because `Jac` is set to the zero matrix before the call to the Jacobian function. The type of `Jac` is `DlsMat`.

The accessor macros `DENSE_ELEM` and `DENSE_COL` allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the `DlsMat` type. `DENSE_ELEM(J, i, j)` references the (`i`, `j`)-th element of the dense matrix `Jac` (`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 `DENSE_ELEM(J, m-1, n-1) =` $J_{m,n}$. Alternatively, `DENSE_COL(J, j)` returns a pointer to the first element of the `j`-th column of `Jac` (`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 = DENSE_COL(J, n-1); col_n[m-1] =` $J_{m,n}$. For large problems, it is more efficient to use `DENSE_COL` than to use `DENSE_ELEM`. Note that both of these macros number rows and columns starting from 0.

The `DlsMat` type and accessor macros `DENSE_ELEM` and `DENSE_COL` are documented in §10.1.3.

If the user's `CVDenseJacFn` 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 §4.5.8.1. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

For the sake of uniformity, the argument `N` is of type `long int`, even in the case that the Lapack dense solver is to be used.

4.6.6 Jacobian information (direct method with banded Jacobian)

If the direct linear solver with banded treatment of the Jacobian is used (i.e. CVBand or CVLapackBand is called in Step 8 of §4.4), the user may provide a function of type CVDlsBandJacFn defined as follows:

CVDlsBandJacFn

Definition
```
typedef int (*CVBandJacFn)(long int N, long int mupper, long int mlower,
                           realtype t, N_Vector y, N_Vector fy,
                           DlsMat Jac, void *user_data,
                           N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
```

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

Arguments

N	is the problem size.
mlower	
mupper	are the lower and upper half-bandwidths of the Jacobian.
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 band Jacobian matrix (of type DlsMat).
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 CVDlsBandJacFn as temporary storage or work space.

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

Notes A user-supplied band Jacobian function must load the band matrix Jac of type DlsMat with the elements of the Jacobian $J(t, y)$ at the point (t,y). Only nonzero elements need to be loaded into Jac because Jac is initialized to the zero matrix before the call to the Jacobian function.

The accessor macros BAND_ELEM, BAND_COL, and BAND_COL_ELEM allow the user to read and write band matrix elements without making specific references to the underlying representation of the DlsMat type. BAND_ELEM(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 BAND_ELEM(J, m-1, n-1) = $J_{m,n}$. The elements within the band are those with -mupper $\le$ m-n $\le$ mlower. Alternatively, BAND_COL(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 BAND_COL_ELEM(col_j, i, j), counting from 0. Thus, for (m, n) within the band, $J_{m,n}$ can be loaded by setting col_n = BAND_COL(J, n-1); BAND_COL_ELEM(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 DlsMat. The array col_n can be indexed from $-$mupper to mlower. For large problems, it is more efficient to use BAND_COL and

BAND_COL_ELEM than to use the BAND_ELEM macro. As in the dense case, these macros all number rows and columns starting from 0.

The DlsMat type and the accessor macros BAND_ELEM, BAND_COL and BAND_COL_ELEM are documented in §10.1.4.

If the user's CVBandJacFn 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 §4.5.8.1. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundials_types.h.

For the sake of uniformity, the arguments N, mlower, and mupper are of type long int, even in the case that the Lapack band solver is to be used.

4.6.7 Jacobian information (direct method with sparse Jacobian)

If the direct linear solver with sparse treatment of the Jacobian is used (i.e., CVKLU or CVSuperLUMT is called in Step 8 of §4.4), the user must provide a function of type CVSlsSparseJacFn defined by:

CVSlsSparseJacFn

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

Purpose This function computes the sparse Jacobian $J = \partial f/\partial y$ (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 sparse Jacobian matrix (of type SlsMat).
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 CVSlsSparseJacFn as temporary storage or work space.

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

Notes A user-supplied sparse Jacobian function must load the compressed-sparse-column 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 row index arrays as needed. The type of Jac is SlsMat, and the amount of allocated space is available within the SlsMat structure as NNZ. The SlsMat type is further documented in the Section §10.2.

If the user's CVSlsSparseJacFn function uses difference quotient approximations to set the specific nonzero matrix entries, 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 §4.5.8.1. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundials_types.h.

4.6.8 Jacobian information (matrix-vector product)

If one of the Krylov iterative linear solvers SPGMR, SPBCG, or SPTFQMR is selected (CVSp* is called in step 8 of §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 to be 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 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 §4.5.8.1. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundials_types.h.

4.6.9 Preconditioning (linear system solution)

If preconditioning is used, then the user must provide a C 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, N_Vector tmp);
```

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 §4.5.8.1).

`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`.

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

4.6.10 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 C 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,
                                  N_Vector tmp1, N_Vector tmp2,
                                  N_Vector tmp3);
```

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

Arguments The arguments of a `CVSpilsPrecSetupFn` are as follows:

`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 = FALSE` means that the Jacobian-related data must be recomputed from scratch. `jok = TRUE` 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 = TRUE` can only occur after a call with `jok = FALSE`.

`jcurPtr` is a pointer to a flag which should be set to `TRUE` if Jacobian data was recomputed, or set to `FALSE` 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`.

`tmp1`

`tmp2`

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

Return value The value to be 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 §4.5.8.1. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

4.7 Preconditioner modules

The efficiency of Krylov iterative methods for the solution of linear systems can be greatly enhanced through preconditioning. For problems in which the user cannot define a more effective, problem-specific preconditioner, CVODE provides a banded preconditioner in the module CVBANDPRE and a band-block-diagonal preconditioner module CVBBDPRE.

4.7.1 A serial banded preconditioner module

This preconditioner provides a band matrix preconditioner for use with any of the Krylov iterative linear solvers, in a serial setting. It uses difference quotients of the ODE right-hand side function `f` to generate a band matrix of bandwidth $m_l + m_u + 1$, where the number of super-diagonals (m_u, the upper half-bandwidth) and sub-diagonals (m_l, the lower half-bandwidth) are specified by the user, and uses this to form a preconditioner for use with the Krylov linear solver. Although this matrix is intended to approximate the Jacobian $\partial f/\partial y$, it may be a very crude approximation. The true Jacobian need not be banded, or its true bandwidth may be larger than $m_l + m_u + 1$, as long as the banded approximation generated here is sufficiently accurate to speed convergence as a preconditioner.

In order to use the CVBANDPRE module, the user need not define any additional functions. Aside from the header files required for the integration of the ODE problem (see §4.3), to use the CVBANDPRE module, the main program must include the header file `cvode_bandpre.h` which declares the needed function prototypes. The following is a summary of the usage of this module. Steps that are unchanged from the skeleton program presented in §4.4 are grayed out.

1. **Set problem dimensions**
2. **Set vector of initial values**
3. **Create CVODE object**

4. **Allocate internal memory**

5. **Set optional inputs**

6. **Attach iterative linear solver, one of:**

   (a) `flag = CVSpgmr(cvode_mem, pretype, maxl);`

   (b) `flag = CVSpbcg(cvode_mem, pretype, maxl);`

   (c) `flag = CVSptfqmr(cvode_mem, pretype, maxl);`

7. **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.

8. **Set linear solver optional inputs**

   Note that the user should not overwrite the preconditioner setup function or solve function through calls to `CVSpilsSet**` optional input functions.

9. **Advance solution in time**

10. **Get optional outputs**

    Additional optional outputs associated with CVBANDPRE are available by way of two routines described below, `CVBandPrecGetWorkSpace` and `CVBandPrecGetNumRhsEvals`.

11. **Deallocate memory for solution vector**

12. **Free solver memory**

The CVBANDPRE preconditioner module is initialized and attached by calling the following function:

`CVBandPrecInit`

Call `flag = CVBandPrecInit(cvode_mem, N, mu, ml);`

Description The function `CVBandPrecInit` initializes the CVBANDPRE preconditioner and allocates required (internal) memory for it.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODE memory block.
- `N` (`long int`) problem dimension.
- `mu` (`long int`) upper half-bandwidth of the Jacobian approximation.
- `ml` (`long int`) 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 CVODE memory block.
`lenrwBP` (`long int`) the number of `realtype` values in the CVBANDPRE workspace.
`leniwBP` (`long int`) the number of integer values in the CVBANDPRE workspace.

Return value The return value `flag` (of type `int`) is one of:
`CVSPILS_SUCCESS` The optional output values have been successfully set.
`CVSPILS_PMEM_NULL` The CVBANDPRE preconditioner has not been initialized.

Notes In terms of problem size N and `smu` $= \min(N-1, \texttt{mu+ml})$, the actual size of the real workspace is $(2\,\texttt{ml} + \texttt{mu} + \texttt{smu} + 2)\,N$ `realtype` words, and the actual size of the integer workspace is N integer words.

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

`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 finite difference banded Jacobian approximation used within the preconditioner setup function.

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

Return value The return value `flag` (of type `int`) is one of:
`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 `CVSpils***GetNumRhsEvals`, and also from `nfevals`, returned by `CVodeGetNumRhsEvals`. The total number of right-hand side function evaluations is the sum of all three of these counters.

4.7.2 A parallel band-block-diagonal preconditioner module

A principal reason for using a parallel ODE solver such as CVODE lies in the solution of partial differential equations (PDEs). Moreover, the use of a Krylov iterative method for the solution of many such problems is motivated by the nature of the underlying linear system of equations (2.4) that must be solved at each time step. The linear algebraic system is large, sparse and structured. However, if a Krylov iterative method is to be effective in this setting, then a nontrivial preconditioner needs to be used. Otherwise, the rate of convergence of the Krylov iterative method is usually unacceptably slow. Unfortunately, an effective preconditioner tends to be problem-specific.

However, we have developed one type of preconditioner that treats a rather broad class of PDE-based problems. It has been successfully used for several realistic, large-scale problems [21] and is included in a software module within the CVODE package. This module works with the parallel vector module NVECTOR_PARALLEL and is usable with any of the Krylov iterative linear solvers. 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)(long int 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 CVODE will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CVode` returns `CV_LSETUP_FAIL`).

Notes 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)(long int Nlocal, realtype t,
                        N_Vector y, void *user_data);
```

Purpose This `cfn` function performs all interprocess communication necessary for the execution of the `gloc` function above, using the input vector `y`.

Arguments
`Nlocal` is the local vector length.
`t` is the value of the independent variable.
`y` is the dependent variable.
`user_data` is a pointer to user data, the same as the `user_data` parameter passed to `CVodeSetUserData`.

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

Notes The `cfn` function is expected to save communicated data in space defined within the data structure `user_data`.

Each call to the `cfn` function is preceded by a call to the right-hand side function `f` with the same (`t`, `y`) arguments. Thus, `cfn` can omit any communication done by `f` if relevant to the evaluation of `glocal`. If all necessary communication was done in `f`, then `cfn = NULL` can be passed in the call to `CVBBDPrecInit` (see below).

Besides the header files required for the integration of the ODE problem (see §4.3), to use the CVBBDPRE module, the main program must include the header file `cvode_bbdpre.h` which declares the needed function prototypes.

The following is a summary of the proper usage of this module. Steps that are unchanged from the skeleton program presented in §4.4 are grayed out.

1. **Initialize MPI**
2. **Set problem dimensions**
3. **Set vector of initial values**
4. **Create CVODE object**
5. **Allocate internal memory**
6. **Set optional inputs**
7. **Attach iterative linear solver, one of:**
   (a) `flag = CVSpgmr(cvode_mem, pretype, maxl);`
   (b) `flag = CVSpbcg(cvode_mem, pretype, maxl);`
   (c) `flag = CVSptfqmr(cvode_mem, pretype, maxl);`

8. **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.

9. **Set linear solver optional inputs**

   Note that the user should not overwrite the preconditioner setup function or solve function through calls to CVSPILS optional input functions.

10. **Advance solution in time**

11. **Get optional outputs**

    Additional optional outputs associated with CVBBDPRE are available by way of two routines described below, `CVBBDPrecGetWorkSpace` and `CVBBDPrecGetNumGfnEvals`.

12. **Deallocate memory for solution vector**

13. **Free solver memory**

14. **Finalize MPI**

The user-callable functions that initialize (step 8 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 CVODE memory block.
`local_N`	(`long int`) local vector length.
`mudq`	(`long int`) upper half-bandwidth to be used in the difference quotient Jacobian approximation.
`mldq`	(`long int`) lower half-bandwidth to be used in the difference quotient Jacobian approximation.
`mukeep`	(`long int`) upper half-bandwidth of the retained banded approximate Jacobian block.
`mlkeep`	(`long int`) 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 CVODE for a subsequent problem, a call to `CVBBDPrecReInit` can be made to change any of the following: the half-bandwidths `mudq` and `mldq` used in the difference-quotient Jacobian approximations, the relative increment `dqrely`, or one of the user-supplied functions `gloc` and `cfn`. If there is a change in any of the linear solver inputs, an additional call to `CVSpgmr`, `CVSpbcg`, or `CVSptfqmr`, and/or one or more of the corresponding `CVSpils***Set***` functions, must also be made (in the proper order).

`CVBBDPrecReInit`

Call `flag = CVBBDPrecReInit(cvode_mem, mudq, mldq, dqrely);`

Description The function `CVBBDPrecReInit` re-initializes the CVBBDPRE preconditioner.

Arguments

`cvode_mem`	(`void *`) pointer to the CVODE memory block.
`mudq`	(`long int`) upper half-bandwidth to be used in the difference quotient Jacobian approximation.
`mldq`	(`long int`) 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 CVODE memory block.
`lenrwBBDP`	(`long int`) local number of `realtype` values in the CVBBDPRE workspace.

`leniwBBDP` (`long int`) local number of integer values in the CVBBDPRE workspace.

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

`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 In terms of `local_N` and `smu` = min(`local_N - 1, mukeep` + `mlkeep`), the actual size of the real workspace is (2 `mlkeep` + `mukeep` + `smu` +2) `local_N` `realtype` words, and the actual size of the integer workspace is `local_N` integer words. These values are local to each process.

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

`CVBBDPrecGetNumGfnEvals`

Call `flag = CVBBDPrecGetNumGfnEvals(cvode_mem, &ngevalsBBDP);`

Description The function `CVBBDPrecGetNumGfnEvals` returns the number of calls made to the user-supplied `gloc` function due to the finite difference approximation of the Jacobian blocks used within the preconditioner setup function.

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

`ngevalsBBDP` (`long int`) the number of calls made to the user-supplied `gloc` function.

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

`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 CVODE output and `npsolves` and `nfevalsLS` are linear solver optional outputs (see §4.5.8).

Chapter 5

FCVODE, an Interface Module for FORTRAN Applications

The FCVODE interface module is a package of C functions which support the use of the CVODE solver, for the solution of ODE systems $dy/dt = f(t, y)$, in a mixed FORTRAN/C setting. While CVODE is written in C, it is assumed here that the user's calling program and user-supplied problem-defining routines are written in FORTRAN. This package provides the necessary interface to CVODE for all supplied serial and parallel NVECTOR implementations.

5.1 Important note on portability

In this package, the names of the interface functions, and the names of the FORTRAN user routines called by them, appear as dummy names which are mapped to actual values by a series of definitions in the header files. By default, those mapping definitions depend in turn on the C macro `F77_FUNC` defined in the header file `sundials_config.h`. The mapping defined by `F77_FUNC` in turn transforms the C interface names to match the name-mangling approach used by the supplied Fortran compiler.

By "name-mangling", we mean that due to the case-independent nature of the FORTRAN language, FORTRAN compilers convert all subroutine and object names to use either all lower-case or all upper-case characters, and append either zero, one or two underscores as a prefix or suffix to the name. For example, the FORTRAN subroutine `MyFunction()` will be changed to one of `myfunction`, `MYFUNCTION`, `myfunction__`, `MYFUNCTION_`, and so on, depending on the FORTRAN compiler used.

SUNDIALS determines this name-mangling scheme at configuration time (see Appendix A).

5.2 Fortran Data Types

Throughout this documentation, we will refer to data types according to their usage in C. The equivalent types to these may vary, depending on your computer architecture and on how SUNDIALS was compiled (see Appendix A). A FORTRAN user should first determine the equivalent types for their architecture and compiler, and then take care that all arguments passed through this FORTRAN/C interface are declared of the appropriate type.

Integers: SUNDIALS uses both `int` and `long int` types:

- `int` – equivalent to an `INTEGER` or `INTEGER*4` in FORTRAN
- `long int` – this will depend on the computer architecture:
  - 32-bit architecture – equivalent to an `INTEGER` or `INTEGER*4` in FORTRAN
  - 64-bit architecture – equivalent to an `INTEGER*8` in FORTRAN

Real numbers: As discussed in Appendix A, at compilation SUNDIALS allows the configuration option `--with-precision`, that accepts values of `single`, `double` or `extended` (the default is `double`). This choice dictates the size of a `realtype` variable. The corresponding FORTRAN types for these `realtype` sizes are:

- `single` – equivalent to a `REAL` or `REAL*4` in FORTRAN
- `double` – equivalent to a `DOUBLE PRECISION` or `REAL*8` in FORTRAN
- `extended` – equivalent to a `REAL*16` in FORTRAN

5.3 FCVODE routines

The user-callable functions, with the corresponding CVODE functions, are as follows:

- Interface to the NVECTOR modules
  - `FNVINITS` (defined by NVECTOR_SERIAL) interfaces to `N_VNewEmpty_Serial`.
  - `FNVINITP` (defined by NVECTOR_PARALLEL) interfaces to `N_VNewEmpty_Parallel`.
  - `FNVINITOMP` (defined by NVECTOR_OPENMP) interfaces to `N_VNewEmpty_OpenMP`.
  - `FNVINITPTS` (defined by NVECTOR_PTHREADS) interfaces to `N_VNewEmpty_Pthreads`.
- Interface to the main CVODE module
  - `FCVMALLOC` interfaces to `CVodeCreate`, `CVodeSetUserData`, and `CVodeInit`, as well as one of `CVodeSStolerances` or `CVodeSVtolerances`.
  - `FCVREINIT` interfaces to `CVodeReInit`.
  - `FCVSETIIN` and `FCVSETRIN` interface to `CVodeSet*` functions.
  - `FCVEWTSET` interfaces to `CVodeWFtolerances`.
  - `FCVODE` interfaces to `CVode`, `CVodeGet*` functions, and to the optional output functions for the selected linear solver module.
  - `FCVDKY` interfaces to the interpolated output function `CVodeGetDky`.
  - `FCVGETERRWEIGHTS` interfaces to `CVodeGetErrWeights`.
  - `FCVGETESTLOCALERR` interfaces to `CVodeGetEstLocalErrors`.
  - `FCVFREE` interfaces to `CVodeFree`.
- Interface to the linear solver modules
  - `FCVDIAG` interfaces to `CVDiag`.
  - `FCVDENSE` interfaces to `CVDense`.
  - `FCVDENSESETJAC` interfaces to `CVDlsSetDenseJacFn`.
  - `FCVLAPACKDENSE` interfaces to `CVLapackDense`.
  - `FCVLAPACKDENSESETJAC` interfaces to `CVDlsSetDenseJacFn`.
  - `FCVBAND` interfaces to `CVBand`.
  - `FCVBANDSETJAC` interfaces to `CVDlsSetBandJacFn`.
  - `FCVLAPACKBAND` interfaces to `CVLapackBand`.
  - `FCVLAPACKBANDSETJAC` interfaces to `CVDlsSetBandJacFn`.
  - `FCVKLU` interfaces to `CVKLU`.
  - `FCVKLUREINIT` interfaces to `CVKLUReInit`.
  - `FCVSUPERLUMT` interfaces to `CVSuperLUMT`.

- FCVSPARSESETJAC interfaces to CVSlsSetSparseJacFn.
- FCVSPGMR interfaces to CVSpgmr and SPGMR optional input functions.
- FCVSPGMRREINIT interfaces to SPGMR optional input functions.
- FCVSPBCG interfaces to CVSpbcg and SPBCG optional input functions.
- FCVSPBCGREINIT interfaces to SPBCG optional input functions.
- FCVSPTFQMR interfaces to CVSptfqmr and SPTFQMR optional input functions.
- FCVSPTFQMRREINIT interfaces to SPTFQMR optional input functions.
- FCVSPILSSETJAC interfaces to CVSpilsSetJacTimesVecFn.
- FCVSPILSSETPREC interfaces to CVSpilsSetPreconditioner.

The user-supplied functions, each listed with the corresponding internal interface function which calls it (and its type within CVODE), are as follows:

FCVODE routine (FORTRAN, user-supplied)	CVODE function (C, interface)	CVODE type of interface function
FCVFUN	FCVf	CVRhsFn
FCVEWT	FCVEwtSet	CVEwtFn
FCVDJAC	FCVDenseJac	CVDlsDenseJacFn
	FCVLapackDenseJac	CVDlsDenseJacFn
FCVBJAC	FCVBandJac	CVDlsBandJacFn
	FCVLapackBandJac	CVDlsBandJacFn
FCVSPJAC	FCVSparseJac	CVSlsSparseJacFn
FCVPSOL	FCVPSol	CVSpilsPrecSolveFn
FCVPSET	FCVPSet	CVSpilsPrecSetupFn
FCVJTIMES	FCVJtimes	CVSpilsJacTimesVecFn

In contrast to the case of direct use of CVODE, and of most FORTRAN ODE solvers, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program.

5.4 Usage of the FCVODE interface module

The usage of FCVODE requires calls to six or seven interface functions, depending on the method options selected, and one or more user-supplied routines which define the problem to be solved. These function calls and user routines are summarized separately below. Some details are omitted, and the user is referred to the description of the corresponding CVODE functions for information on the arguments of any given user-callable interface routine, or of a given user-supplied function called by an interface function. The usage of FCVODE for rootfinding and with preconditioner modules is described in later subsections.

1. **Right-hand side specification**

   The user must, in all cases, supply the following FORTRAN routine

   ```
   SUBROUTINE FCVFUN(T, Y, YDOT, IPAR, RPAR, IER)
   DIMENSION Y(*), YDOT(*), IPAR(*), RPAR(*)
   ```

   It must set the YDOT array to $f(t, y)$, the right-hand side of the ODE system, as function of T= t and the array Y= y. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted).

2. NVECTOR **module initialization**

   If using one of the NVECTOR modules supplied with SUNDIALS, the user must make a call of the form

   ```
   CALL FNVINIT***(...)
   ```

   in which the name and call sequence are as described in the appropriate section of Chapter 6.

3. **Problem specification**

   To set various problem and solution parameters and allocate internal memory, make the following call:

   `FCVMALLOC`

   Call
   ```
   CALL FCVMALLOC(T0, Y0, METH, ITMETH, IATOL, RTOL, ATOL,
   &              IOUT, ROUT, IPAR, RPAR, IER)
   ```

   Description: This function provides required problem and solution specifications, specifies optional inputs, allocates internal memory, and initializes CVODE.

   Arguments:
   - `T0` is the initial value of t.
   - `Y0` is an array of initial conditions.
   - `METH` specifies the basic integration method: 1 for Adams (nonstiff) or 2 for BDF (stiff).
   - `ITMETH` specifies the nonlinear iteration method: 1 for functional iteration or 2 for Newton iteration.
   - `IATOL` specifies the type for absolute tolerance `ATOL`: 1 for scalar or 2 for array. If `IATOL`= 3, the arguments `RTOL` and `ATOL` are ignored and the user is expected to subsequently call `FCVEWTSET` and provide the function `FCVEWT`.
   - `RTOL` is the relative tolerance (scalar).
   - `ATOL` is the absolute tolerance (scalar or array).
   - `IOUT` is an integer array of length 21 for integer optional outputs.
   - `ROUT` is a real array of length 6 for real optional outputs.
   - `IPAR` is an integer array of user data which will be passed unmodified to all user-provided routines.
   - `RPAR` is a real array of user data which will be passed unmodified to all user-provided routines.

   Return value: `IER` is a return completion flag. Values are 0 for successful return and -1 otherwise. See printed message for details in case of failure.

   Notes: The user integer data arrays `IOUT` and `IPAR` must be declared as `INTEGER*4` or `INTEGER*8` according to the C type `long int`.

   Modifications to the user data arrays `IPAR` and `RPAR` inside a user-provided routine will be propagated to all subsequent calls to such routines.

   The optional outputs associated with the main CVODE integrator are listed in Table 5.2.

As an alternative to providing tolerances in the call to `FCVMALLOC`, the user may provide a routine to compute the error weights used in the WRMS norm evaluations. If supplied, it must have the following form:

```
SUBROUTINE FCVEWT (Y, EWT, IPAR, RPAR, IER)
DIMENSION Y(*), EWT(*), IPAR(*), RPAR(*)
```

It must set the positive components of the error weight vector `EWT` for the calculation of the WRMS norm of `Y`. On return, set `IER = 0` if `FCVEWT` was successful, and nonzero otherwise. The

arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC.

If the FCVEWT routine is provided, then, following the call to FCVMALOC, the user must make the call:

```
CALL FCVEWTSET (FLAG, IER)
```

with FLAG $\neq 0$ to specify use of the user-supplied error weight routine. The argument IER is an error return flag which is 0 for success or non-zero if an error occurred.

4. **Set optional inputs**

   Call FCVINSETIIN and/or FCVINSETRIN to set desired optional inputs, if any. See §5.5 for details.

5. **Linear solver specification**

   In the case of a stiff system, the implicit BDF method involves the solution of linear systems related to the Jacobian $J = \partial f/\partial y$ of the ODE system. The user of FCVODE must call a routine with a specific name to make the desired choice of linear solver. Note that the direct (dense or band) and sparse linear solver options are usable only in a serial environment.

   Diagonal approximate Jacobian

   This choice is appropriate when the Jacobian can be well approximated by a diagonal matrix. The user must make the call:

   ```
   CALL FCVDIAG(IER)
   ```

   IER is an error return flag set on 0 on success or -1 if a memory failure occurred. There is no additional user-supplied routine. Optional outputs specific to the DIAG case listed in Table 5.2.

   Dense treatment of the linear system

   To use the direct dense linear solver based on the internal CVODE implementation, the user must make the call:

   ```
   CALL FCVDENSE(NEQ, IER)
   ```

   where NEQ is the size of the ODE system. The argument IER is an error return flag which is 0 for success , -1 if a memory allocation failure occurred, or -2 for illegal input.

   Alternatively, to use the Lapack-based direct dense linear solver, the user must make the call:

   ```
   CALL FCVLAPACKDENSE(NEQ, IER)
   ```

   where the arguments have the same meanings as for FCVDENSE, except that here NEQ must be declared so as to match C type int.

   As an option when using the DENSE linear solver, the user may supply a routine that computes a dense approximation of the system Jacobian $J = \partial f/\partial y$. If supplied, it must have the following form:

   ```
    SUBROUTINE FCVDJAC (NEQ, T, Y, FY, DJAC, H, IPAR, RPAR,
   &                    WK1, WK2, WK3, IER)
    DIMENSION Y(*), FY(*), DJAC(NEQ,*), IPAR(*), RPAR(*),
   &          WK1(*), WK2(*), WK3(*)
   ```

   Typically this routine will use only NEQ, T, Y, and DJAC. It must compute the Jacobian and store it columnwise in DJAC. The input arguments T, Y, and FY contain the current values of t, y, and

$f(t, y)$, respectively. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. The vectors WK1, WK2, and WK3 of length NEQ are provided as work space for use in FCVDJAC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if FCVDJAC failed unrecoverably (in which case the integration is halted). NOTE: The argument NEQ has a type consistent with C type `long int` even in the case when the Lapack dense solver is to be used.

If the user's FCVDJAC uses difference quotient approximations, it may need to use the error weight array EWT and current stepsize H in the calculation of suitable increments. The array EWT can be obtained by calling FCVGETERRWEIGHTS using one of the work arrays as temporary storage for EWT. It may also need the unit roundoff, which can be obtained as the optional output ROUT(6), passed from the calling program to this routine using either RPAR or a common block.

If the FCVDJAC routine is provided, then, following the call to FCVDENSE, the user must make the call:

```
CALL FCVDENSESETJAC (FLAG, IER)
```

with FLAG $\neq 0$ to specify use of the user-supplied Jacobian approximation. The argument IER is an error return flag which is 0 for success or non-zero if an error occurred. If using the Lapack-based direct dense linear solver, the use of a Jacobian approximation supplied by the user is indicated through the call

```
CALL FCVLAPACKDENSESETJAC (FLAG, IER)
```

Optional outputs specific to the DENSE case are listed in Table 5.2.

Band treatment of the linear system

To use the direct band linear solver based on the internal CVODE implementation, the user must make the call:

```
CALL FCVBAND (NEQ, MU, ML, IER)
```

The arguments are: MU, the upper half-bandwidth; ML, the lower half-bandwidth; and IER an error return flag which is 0 for success , -1 if a memory allocation failure occurred, or -2 in case an input has an illegal value.

Alternatively, to use the Lapack-based direct band linear solver, the user must make the call:

```
CALL FCVLAPACKBAND(NEQ, MU, ML, IER)
```

where the arguments have the same meanings as for FCVBAND, except that here NEQ, MU, and ML must be declared so as to match C type `int`.

As an option when using the BAND linear solver, the user may supply a routine that computes a band approximation of the system Jacobian $J = \partial f/\partial y$. If supplied, it must have the following form:

```
 SUBROUTINE FCVBJAC(NEQ, MU, ML, MDIM, T, Y, FY, BJAC, H, IPAR, RPAR,
&                   WK1, WK2, WK3, IER)
 DIMENSION Y(*), FY(*), BJAC(MDIM,*), IPAR(*), RPAR(*),
&               WK1(*), WK2(*), WK3(*)
```

Typically this routine will use only NEQ, MU, ML, T, Y, and BJAC. It must load the MDIM by N array BJAC with the Jacobian matrix at the current (t,y) in band form. Store in BJAC(k, j) the

Jacobian element $J_{i,j}$ with $k = i - j +$ MU $+1$ ($k = 1 \cdots$ ML + MU + 1) and $j = 1 \cdots N$. The input arguments T, Y, and FY contain the current values of t, y, and $f(t, y)$, respectively. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. The vectors WK1, WK2, and WK3 of length NEQ are provided as work space for use in FCVBJAC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if FCVBJAC failed unrecoverably (in which case the integration is halted). NOTE: The arguments NEQ, MU, ML, and MDIM have a type consistent with C type long int even in the case when the Lapack band solver is to be used.

If the user's FCVBJAC uses difference quotient approximations, it may need to use the error weight array EWT and current stepsize H in the calculation of suitable increments. The array EWT can be obtained by calling FCVGETERRWEIGHTS using one of the work arrays as temporary storage for EWT. It may also need the unit roundoff, which can be obtained as the optional output ROUT(6), passed from the calling program to this routine using either RPAR or a common block.

If the FCVBJAC routine is provided, then, following the call to FCVBAND, the user must make the call:

```
CALL FCVBANDSETJAC(FLAG, IER)
```

with FLAG $\neq 0$ to specify use of the user-supplied Jacobian approximation. The argument IER is an error return flag which is 0 for success or non-zero if an error occurred. If using the Lapack-based direct band linear solver, the use of a Jacobian approximation supplied by the user is indicated through the call

```
CALL FCVLAPACKNBANDSETJAC (FLAG, IER)
```

Optional outputs specific to the BAND case are listed in Table 5.2.

Sparse direct treatment of the linear system

To use the KLU sparse direct linear solver, the user must make the call:

```
CALL FCVKLU (NEQ, NNZ, SPARSETYPE, ORDERING, IER)
```

where NEQ is the size of the ODE system, NNZ is the maximum number of nonzeros in the Jacobian matrix, SPARSETYPE is a flag indicating whether the matrix is in compressed-sparse-column or compressed-sparse-row format (0 = CSC, 1 = CSR), and ORDERING is the matrix ordering desired with possible values from the KLU package (0 = AMD, 1 = COLAMD). The argument IER is an error return flag which is 0 for success or negative for an error.

The CVODE KLU solver will reuse much of the factorization information from one nonlinear iteration and time step to the next. If at any time the user wants to force a full refactorization, or if the number of nonzeros in the Jacobian matrix changes, the user should make the call

```
CALL FCVKLUREINIT(NEQ, NNZ, REINIT_TYPE)
```

where NEQ is the size of the ODE system, NNZ is the maximum number of nonzeros in the Jacobian matrix, and REINIT_TYPE is 1 or 2. For a value of 1, the matrix will be destroyed and a new one will be allocated with NNZ nonzeros. For a value of 2, only symbolic and numeric factorizations will be completed.

Alternatively, to use the SuperLUMT linear solver, the user must make the call:

```
CALL FCVSUPERLUMT (NTHREADS, NEQ, NNZ, ORDERING, IER)
```

where NTHREADS is the number of threads requested and the other arguments have the same meanings as for FCVKLU, except that here possible values for ORDERING derive from the SUPERLUMT package and include: 0 for Natural ordering, 1 for Minimum degree on A^TA, 2 for Minimum degree on $A^T + A$, and 3 for COLAMD.

If the either of the sparse direct interface packages are used, then the user must supply the FCVSPJAC routine that computes a compressed-sparse-column [or compressed-sparse-row if using KLU] approximation of the system Jacobian $J = \partial f/\partial y$. If supplied, it must have the following form:

```
 SUBROUTINE FCVSPJAC(T, Y, FY, N, NNZ, JDATA, JRVALS,
&                    JCPTRS, H, IPAR, RPAR, WK1, WK2, WK3, IER)
```

It must load the N by N compressed sparse column [or compressed sparse row] matrix with storage for NNZ nonzeros, stored in the arrays JDATA (nonzero values), JRVALS (row [or column] indices for each nonzero), JCOLPTRS (indices for start of each column [or row]), with the Jacobian matrix at the current (t, y) in CSC [or CSR] form (see sundials_sparse.h for more information). The arguments are T, the current time; Y, an array containing state variables; FY, an array containing state derivatives; N, the number of matrix rows/columns in the Jacobian; NNZ, allocated length of nonzero storage; JDATA, nonzero values in the Jacobian (of length NNZ); JRVALS, row [or column] indices for each nonzero in Jacobian (of length NNZ); JCPTRS, pointers to each Jacobian column [or row] in the two preceding arrays (of length N+1); H, the current step size; IPAR, an array containing integer user data that was passed to FCVMALLOC; RPAR, an array containing real user data that was passed to FCVMALLOC; WK*, work arrays containing temporary workspace of same size as Y; and IER, error return code (0 if successful, > 0 if a recoverable error occurred, or < 0 if an unrecoverable error occurred.)

To indicate that the FCVSPJAC routine has been provided, then following either the call to FCVKLU or FCVSUPERLUMT, the following call must be made

```
CALL FCVSPARSESETJAC (IER)
```

The int return flag IER is an error return flag which is 0 for success or nonzero for an error.

Optional outputs specific to the SPARSE case are listed in Table 5.2.

SPGMR treatment of the linear systems

For the Scaled Preconditioned GMRES solution of the linear systems, the user must make the call

```
CALL FCVSPGMR(IPRETYPE, IGSTYPE, MAXL, DELT, IER)
```

The arguments are as follows. IPRETYPE specifies the preconditioner type: 0 for no preconditioning, 1 for left only, 2 for right only, or 3 for both sides. IGSTYPE indicates the Gram-Schmidt process type: 1 for modified G-S or 2 for classical G-S. MAXL is the maximum Krylov subspace dimension. DELT is the linear convergence tolerance factor. For all of the input arguments, a value of 0 or 0.0 indicates the default. IER is an error return flag which is 0 to indicate success, -1 if a memory allocation failure occurred, or -2 to indicate an illegal input.

Optional outputs specific to the SPGMR case are listed in Table 5.2.

For descriptions of the relevant optional user-supplied routines, see **User-supplied routines for SPGMR/SPBCG/SPTFQMR** below.

SPBCG treatment of the linear systems

For the Scaled Preconditioned Bi-CGStab solution of the linear systems, the user must make the call

```
CALL FCVSPBCG(IPRETYPE, MAXL, DELT, IER)
```

Its arguments are the same as those with the same names for FCVSPGMR.

Optional outputs specific to the SPBCG case are listed in Table 5.2.

For descriptions of the relevant optional user-supplied routines, see **User-supplied routines for SPGMR/SPBCG/SPTFQMR** below.

SPTFQMR treatment of the linear systems

For the Scaled Preconditioned Transpose-Free Quasi-Minimal Residual solution of the linear systems, the user must make the call

```
CALL FCVSPTFQMR(IPRETYPE, MAXL, DELT, IER)
```

Its arguments are the same as those with the same names for FCVSPGMR.

Optional outputs specific to the SPTFQMR case are listed in Table 5.2.

For descriptions of the relevant optional user-supplied routines, see below.

Functions used by SPGMR/SPBCG/SPTFQMR

An optional user-supplied routine, FCVJTIMES (see below), can be provided for Jacobian-vector products. If it is, then, following the call to FCVSPGMR, FCVSPBCG, or FCVSPTFQMR, the user must make the call:

```
CALL FCVSPILSSETJAC(FLAG, IER)
```

with FLAG $\neq 0$ to specify use of the user-supplied Jacobian-times-vector approximation. The argument IER is an error return flag which is 0 for success or non-zero if an error occurred.

If preconditioning is to be done (IPRETYPE $\neq 0$), then the user must call

```
CALL FCVSPILSSETPREC(FLAG, IER)
```

with FLAG $\neq 0$. The return flag IER is 0 if successful, or negative if a memory error occurred. In addition, the user program must include preconditioner routines FCVPSOL and FCVPSET (see below).

User-supplied routines for SPGMR/SPBCG/SPTFQMR

With treatment of the linear systems by any of the Krylov iterative solvers, there are three optional user-supplied routines — FCVJTIMES, FCVPSOL, and FCVPSET. The specifications for these routines are given below.

As an option when using the SPGMR, SPBCG, or SPTFQMR linear solvers, the user may supply a routine that computes the product of the system Jacobian $J = \partial f/\partial y$ and a given vector v. If supplied, it must have the following form:

```
SUBROUTINE FCVJTIMES (V, FJV, T, Y, FY, H, IPAR, RPAR, WORK, IER)
DIMENSION V(*), FJV(*), Y(*), FY(*), IPAR(*), RPAR(*), WORK(*)
```

Typically this routine will use only NEQ, T, Y, V, and FJV. It must compute the product vector Jv, where the vector v is stored in V, and store the product in FJV. The input arguments T, Y, and FY contain the current values of t, y, and $f(t, y)$, respectively. On return, set IER = 0 if FCVJTIMES was successful, and nonzero otherwise. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. The vector WORK, of length NEQ, is provided as work space for use in FCVJTIMES.

If preconditioning is to be included, the following routine must be supplied, for solution of the preconditioner linear system:

```
      SUBROUTINE FCVPSOL(T, Y, FY, R, Z, GAMMA, DELTA, LR, IPAR, RPAR,
     &                   WORK, IER)
      DIMENSION Y(*), FY(*), R(*), Z(*), IPAR(*), RPAR(*), WORK(*)
```

It must solve the preconditioner linear system $Pz = r$, where r = R is input, and store the solution z in Z. Here P is the left preconditioner if LR=1 and the right preconditioner if LR=2. The preconditioner (or the product of the left and right preconditioners if both are nontrivial) should be an approximation to the matrix $I - \gamma J$, where I is the identity matrix, J is the system Jacobian, and γ = GAMMA. The input arguments T, Y, and FY contain the current values of t, y, and $f(t, y)$, respectively. On return, set IER = 0 if FCVPSOL was successful, set IER positive if a recoverable error occurred, and set IER negative if a non-recoverable error occurred.

The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. The argument WORK is a work array of length NEQ for use by this routine.

If the user's preconditioner requires that any Jacobian related data be evaluated or preprocessed, then the following routine can be used for the evaluation and preprocessing of the preconditioner:

```
      SUBROUTINE FCVPSET(T, Y, FY, JOK, JCUR, GAMMA, H, IPAR, RPAR,
     &                   WORK1, WORK2, WORK3, IER)
      DIMENSION Y(*), FY(*), EWT(*), IPAR(*), RPAR(*),
     &          WORK1(*), WORK2(*), WORK3(*)
```

It must perform any evaluation of Jacobian-related data and preprocessing needed for the solution of the preconditioner linear systems by FCVPSOL. The input argument JOK allows for Jacobian data to be saved and reused: If JOK = 0, this data should be recomputed from scratch. If JOK = 1, a saved copy of it may be reused, and the preconditioner constructed from it. The input arguments T, Y, and FY contain the current values of t, y, and $f(t, y)$, respectively. On return, set JCUR = 1 if Jacobian data was computed, and set JCUR = 0 otherwise. Also on return, set IER = 0 if FCVPSET was successful, set IER positive if a recoverable error occurred, and set IER negative if a non-recoverable error occurred.

The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. The arguments WORK1, WORK2, WORK3 are work arrays of length NEQ for use by this routine.

If the user calls FCVSPILSSETPREC, the routine FCVPSET must be provided, even if it is not needed, and it must return IER=0.

Notes

(a) If the user's FCVJTIMES or FCVPSET routine uses difference quotient approximations, it may need to use the error weight array EWT, the current stepsize H, and/or the unit roundoff, in the calculation of suitable increments. Also, If FCVPSOL uses an iterative method in its solution, the residual vector $\rho = r - Pz$ of the system should be made less than DELTA in weighted ℓ_2 norm, i.e. $\sqrt{\sum(\rho_i * \texttt{EWT}[i])^2} <$ DELTA.

(b) If needed in FCVJTIMES, FCVPSOL, or FCVPSET, the error weight array EWT can be obtained by calling FCVGETERRWEIGHTS using one of the work arrays as temporary storage for EWT.

(c) If needed in FCVJTIMES, FCVPSOL, or FCVPSET, the unit roundoff can be obtained as the optional output ROUT(6) (available after the call to FCVMALLOC) and can be passed using either the RPAR user data array or a common block.

6. **Problem solution**

Carrying out the integration is accomplished by making calls as follows:

```
      CALL FCVODE(TOUT, T, Y, ITASK, IER)
```

The arguments are as follows. TOUT specifies the next value of t at which a solution is desired (input). T is the value of t reached by the solver on output. Y is an array containing the computed solution on output. ITASK is a task indicator and should be set to 1 for normal mode (overshoot TOUT and interpolate), or to 2 for one-step mode (return after each internal step taken). IER is a completion flag and will be set to a positive value upon successful return or to a negative value if an error occurred. These values correspond to the CVode returns (see §4.5.5 and §B.2). The current values of the optional outputs are available in IOUT and ROUT (see Table 5.2).

7. **Additional solution output**

After a successful return from FCVODE, the routine FCVDKY may be used to obtain a derivative of the solution, of order up to the current method order, at any t within the last step taken. For this, make the following call:

```
CALL FCVDKY(T, K, DKY, IER)
```

where T is the value of t at which solution derivative is desired, and K is the derivative order ($0 \leq$ K $\leq$ QU). On return, DKY is an array containing the computed K-th derivative of y. The value T must lie between TCUR - HU and TCUR. The return flag IER is set to 0 upon successful return or to a negative value to indicate an illegal input.

8. **Problem reinitialization**

To re-initialize the CVODE solver for the solution of a new problem of the same size as one already solved, make the following call:

```
CALL FCVREINIT(T0, Y0, IATOL, RTOL, ATOL, IER)
```

The arguments have the same names and meanings as those of FCVMALLOC. FCVREINIT performs the same initializations as FCVMALLOC, but does no memory allocation, using instead the existing internal memory created by the previous FCVMALLOC call. The call to specify the linear system solution method may or may not be needed.

Following this call, a call to specify the linear system solver must be made if the choice of linear solver is being changed. Otherwise, a call to reinitialize the linear solver last used may or may not be needed, depending on changes in the inputs to it.

In the case of the BAND solver, for any change in the half-bandwidth parameters, call FCVBAND (or FCVLAPACKBAND) as described above.

In the case of SPGMR, for a change of inputs other than MAXL, make the call

```
CALL FCVSPGMRREINIT (IPRETYPE, IGSTYPE, DELT, IER)
```

which reinitializes SPGMR without reallocating its memory. The arguments have the same names and meanings as those of FCVSPGMR. If MAXL is being changed, then call FCVSPGMR instead.

In the case of SPBCG, for a change in any inputs, make the call

```
CALL FCVSPBCGREINIT (IPRETYPE, MAXL, DELT, IER)
```

which reinitializes SPBCG without reallocating its memory. The arguments have the same names and meanings as those of FCVSPBCG.

In the case of SPTFQMR, for a change in any inputs, make the call

```
CALL FCVSPTFQMRREINIT (IPRETYPE, MAXL, DELT, IER)
```

which reinitializes SPTFQMR without reallocating its memory. The arguments have the same names and meanings as those of FCVSPTFQMR.

Table 5.1: Keys for setting FCVODE optional inputs

Integer optional inputs (FCVSETIIN)

Key	Optional input	Default value
MAX_ORD	Maximum LMM method order	5 (BDF), 12 (Adams)
MAX_NSTEPS	Maximum no. of internal steps before t_{out}	500
MAX_ERRFAIL	Maximum no. of error test failures	7
MAX_NITERS	Maximum no. of nonlinear iterations	3
MAX_CONVFAIL	Maximum no. of convergence failures	10
HNIL_WARNS	Maximum no. of warnings for $t_n + h = t_n$	10
STAB_LIM	Flag to activate stability limit detection	0

Real optional inputs (FCVSETRIN)

Key	Optional input	Default value
INIT_STEP	Initial step size	estimated
MAX_STEP	Maximum absolute step size	∞
MIN_STEP	Minimum absolute step size	0.0
STOP_TIME	Value of t_{stop}	undefined
NLCONV_COEF	Coefficient in the nonlinear convergence test	0.1

9. **Memory deallocation**

   To free the internal memory created by the call to FCVMALLOC, make the call

```
CALL FCVFREE
```

5.5 FCVODE optional input and output

In order to keep the number of user-callable FCVODE interface routines to a minimum, optional inputs to the CVODE solver are passed through only two routines: FCVSETIIN for integer optional inputs and FCVSETRIN for real optional inputs. These functions should be called as follows:

```
CALL FCVSETIIN(KEY, IVAL, IER)
CALL FCVSETRIN(KEY, RVAL, IER)
```

where KEY is a quoted string indicating which optoinal input is set (see Table 5.1), IVAL is the integer input value to be used, RVAL is the real input value to be used, and IER is an integer return flag which is set to 0 on success and a negative value if a failure occurred. The integer IVAL should be declared in a manner consistent with C type `long int`.

The optional outputs from the CVODE solver are accessed not through individual functions, but rather through a pair of arrays, IOUT (integer type) of dimension at least 21, and ROUT (real type) of dimension at least 6. These arrays are owned (and allocated) by the user and are passed as arguments to FCVMALLOC. Table 5.2 lists the entries in these two arrays and specifies the optional variable as well as the CVODE function which is actually called to extract the optional output.

For more details on the optional inputs and outputs, see §4.5.6 and §4.5.8.

In addition to the optional inputs communicated through FCVSET* calls and the optional outputs extracted from IOUT and ROUT, the following user-callable routines are available:

To obtain the error weight array EWT, containing the multiplicative error weights used the WRMS norms, make the following call:

```
CALL FCVGETERRWEIGHTS (EWT, IER)
```

This computes the EWT array normally defined by Eq. (2.6). The array EWT, of length NEQ or NLOCAL, must already have been declared by the user. The error return flag IER is zero if successful, and negative if there was a memory error.

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

Integer output array IOUT

Index	Optional output	CVODE function
CVODE main solver		
1	LENRW	CVodeGetWorkSpace
2	LENIW	CVodeGetWorkSpace
3	NST	CVodeGetNumSteps
4	NFE	CVodeGetNumRhsEvals
5	NETF	CVodeGetNumErrTestFails
6	NCFN	CVodeGetNumNonlinSolvConvFails
7	NNI	CVodeGetNumNonlinSolvIters
8	NSETUPS	CVodeGetNumLinSolvSetups
9	QU	CVodeGetLastOrder
10	QCUR	CVodeGetCurrentOrder
11	NOR	CVodeGetNumStabLimOrderReds
12	NGE	CVodeGetNumGEvals
CVDLS linear solvers		
13	LENRWLS	CVDlsGetWorkSpace
14	LENIWLS	CVDlsGetWorkSpace
15	LS_FLAG	CVDlsGetLastFlag
16	NFELS	CVDlsGetNumRhsEvals
17	NJE	CVDlsGetNumJacEvals
CVDIAG linear solver		
13	LENRWLS	CVDiagGetWorkSpace
14	LENIWLS	CVDiagGetWorkSpace
15	LS_FLAG	CVDiagGetLastFlag
16	NFELS	CVDiagGetNumRhsEvals
CVSLS linear solvers		
14	LS_FLAG	CVSlsGetLastFlag
16	NJE	CVSlsGetNumJacEvals
CVSPILS linear solvers		
13	LENRWLS	CVSpilsGetWorkSpace
14	LENIWLS	CVSpilsGetWorkSpace
15	LS_FLAG	CVSpilsGetLastFlag
16	NFELS	CVSpilsGetNumRhsEvals
17	NJTV	CVSpilsGetNumJacEvals
18	NPE	CVSpilsGetNumPrecEvals
19	NPS	CVSpilsGetNumPrecSolves
20	NLI	CVSpilsGetNumLinIters
21	NCFL	CVSpilsGetNumConvFails

Real output array ROUT

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

To obtain the estimated local errors, following a successful call to FCVSOLVE, make the following call:

```
CALL FCVGETESTLOCALERR (ELE, IER)
```

This computes the ELE array of estimated local errors as of the last step taken. The array ELE must already have been declared by the user. The error return flag IER is zero if successful, and negative if there was a memory error.

5.6 Usage of the FCVROOT interface to rootfinding

The FCVROOT interface package allows programs written in FORTRAN to use the rootfinding feature of the CVODE solver module. The user-callable functions in FCVROOT, with the corresponding CVODE functions, are as follows:

- FCVROOTINIT interfaces to CVodeRootInit.
- FCVROOTINFO interfaces to CVodeGetRootInfo.
- FCVROOTFREE interfaces to CVodeRootFree.

Note that at this time, FCVROOT does not provide support to specify the direction of zero-crossing that is to be monitored. Instead, all roots are considered. However, the actual direction of zero-crossing is reported (through the sign of the non-zero elements in the array INFO returned by FCVROTINFO).

In order to use the rootfinding feature of CVODE, the following call must be made, after calling FCVMALLOC but prior to calling FCVODE, to allocate and initialize memory for the FCVROOT module:

```
CALL FCVROOTINIT (NRTFN, IER)
```

The arguments are as follows: NRTFN is the number of root functions. IER is a return completion flag; its values are 0 for success, -1 if the CVODE memory was NULL, and -11 if a memory allocation failed.

To specifiy the functions whose roots are to be found, the user must define the following routine:

```
SUBROUTINE FCVROOTFN (T, Y, G, IPAR, RPAR, IER)
DIMENSION Y(*), G(*), IPAR(*), RPAR(*)
```

It must set the G array, of length NRTFN, with components $g_i(t, y)$, as a function of T $= t$ and the array Y $= y$. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. Set IER on 0 if successful, or on a non-zero value if an error occurred.

When making calls to FCVODE to solve the ODE system, the occurrence of a root is flagged by the return value IER $= 2$. In that case, if NRTFN > 1, the functions g_i which were found to have a root can be identified by making the following call:

```
CALL FCVROOTINFO (NRTFN, INFO, IER)
```

The arguments are as follows: NRTFN is the number of root functions. INFO is an integer array of length NRTFN with root information. IER is a return completion flag; its values are 0 for success, negative if there was a memory failure. The returned values of INFO(i) (i$= 1, \ldots,$ NRTFN) are 0 or ± 1, such that INFO(i) $= +1$ if $g_{\mathtt{i}}$ was found to have a root and $g_{\mathtt{i}}$ is increasing, INFO(i) $= -1$ if $g_{\mathtt{i}}$ was found to have a root and $g_{\mathtt{i}}$ is dereasing, and INFO(i) $= 0$ otherwise.

The total number of calls made to the root function FCVROOTFN, denoted NGE, can be obtained from IOUT(12). If the FCVODE/CVODE memory block is reinitialized to solve a different problem via a call to FCVREINIT, then the counter NGE is reset to zero.

To free the memory resources allocated by a prior call to FCVROOTINIT, make the following call:

```
CALL FCVROOTFREE
```

5.7 Usage of the FCVBP interface to CVBANDPRE

The FCVBP interface sub-module is a package of C functions which, as part of the FCVODE interface module, support the use of the CVODE solver with the serial NVECTOR_SERIAL module, and the combination of the CVBANDPRE preconditioner module (see §4.7.1) with any of the Krylov iterative linear solvers.

The two user-callable functions in this package, with the corresponding CVODE function around which they wrap, are:

- `FCVBPINIT` interfaces to `CVBandPrecInit`.
- `FCVBPOPT` interfaces to CVBANDPRE optional output functions.

As with the rest of the FCVODE routines, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file `fcvbp.h`.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in §5.4 are grayed-out.

1. **Right-hand side specification**
2. NVECTOR **module initialization**
3. **Problem specification**
4. **Set optional inputs**
5. **Linear solver specification**

   First, specify one of the CVSPILS iterative linear solvers, by calling one of `FCVSPGMR`, `FCVSPBCG`, or `FCVSPTFQMR`.

   Then, to initialize the CVBANDPRE preconditioner, make the following call:

   ```
   CALL FCVBPINIT(NEQ, MU, ML, IER)
   ```

   The arguments are as follows. `NEQ` is the problem size. `MU` and `ML` are the upper and lower half-bandwidths of the band matrix that is retained as an approximation of the Jacobian. `IER` is a return completion flag. A value of 0 indicates success, while a value of -1 indicates that a memory failure occurred.

   Optionally, to specify that SPGMR, SPBCG, or SPTFQMR should use the supplied `FCVJTIMES`, make the call

   ```
   CALL FCVSPILSSETJAC(FLAG, IER)
   ```

   with `FLAG` $\neq 0$ (see step 5 in §5.4 for details).
6. **Problem solution**
7. CVBANDPRE **Optional outputs**

   Optional outputs specific to the SPGMR, SPBCG, or SPTFQMR solver are listed in Table 5.2. To obtain the optional outputs associated with the CVBANDPRE module, make the following call:

   ```
   CALL FCVBPOPT(LENRWBP, LENIWBP, NFEBP)
   ```

   The arguments should be consistent with C type `long int`. Their returned values are as follows: `LENRWBP` is the length of real preconditioner work space, in `realtype` words. `LENIWBP` is the length of integer preconditioner work space, in integer words. `NFEBP` is the number of $f(t, y)$ evaluations (calls to `FCVFUN`) for difference-quotient banded Jacobian approximations.

8. **Memory deallocation**

   (The memory allocated for the FCVBP module is deallocated automatically by `FCVFREE`.)

5.8 Usage of the FCVBBD interface to CVBBDPRE

The FCVBBD interface sub-module is a package of C functions which, as part of the FCVODE interface module, support the use of the CVODE solver with the parallel NVECTOR_PARALLEL module, and the combination of the CVBBDPRE preconditioner module (see §4.7.2) with any of the Krylov iterative linear solvers.

The user-callable functions in this package, with the corresponding CVODE and CVBBDPRE functions, are as follows:

- `FCVBBDINIT` interfaces to `CVBBDPrecInit`.
- `FCVBBDREINIT` interfaces to `CVBBDPrecReInit`.
- `FCVBBDOPT` interfaces to CVBBDPRE optional output functions.

In addition to the FORTRAN right-hand side function `FCVFUN`, the user-supplied functions used by this package, are listed below, each with the corresponding interface function which calls it (and its type within CVBBDPRE or CVODE):

FCVBBD routine (FORTRAN, user-supplied)	CVODE function (C, interface)	CVODE type of interface function
`FCVLOCFN`	`FCVgloc`	`CVLocalFn`
`FCVCOMMF`	`FCVcfn`	`CVCommFn`
`FCVJTIMES`	`FCVJtimes`	`CVSpilsJacTimesVecFn`

As with the rest of the FCVODE routines, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program. Additionally, based on flags discussed above in §5.3, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file `fcvbbd.h`.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in §5.4 are grayed-out.

1. **Right-hand side specification**
2. **NVECTOR module initialization**
3. **Problem specification**
4. **Set optional inputs**
5. **Linear solver specification**

   First, specify one of the CVSPILS iterative linear solvers, by calling one of `FCVSPGMR`, `FCVSPBCG`, or `FCVSPTFQMR`.

   Then, to initialize the CVBBDPRE preconditioner, make the following call:

   ```
   CALL FCVBBDINIT(NLOCAL, MUDQ, MLDQ, MU, ML, DQRELY, IER)
   ```

   The arguments are as follows. `NLOCAL` is the local size of vectors on this processor. `MUDQ` and `MLDQ` are the upper and lower half-bandwidths to be used in the computation of the local Jacobian blocks by difference quotients. These may be smaller than the true half-bandwidths of the Jacobian of the local block of g, when smaller values may provide greater efficiency. `MU` and `ML` are the upper and lower half-bandwidths of the band matrix that is retained as an approximation of the local

Jacobian block. These may be smaller than MUDQ and MLDQ. DQRELY is the relative increment factor in y for difference quotients (optional). A value of 0.0 indicates the default, $\sqrt{\text{unit roundoff}}$. IER is a return completion flag. A value of 0 indicates success, while a value of -1 indicates that a memory failure occurred or that an input had an illegal value.

Optionally, to specify that SPGMR, SPBCG, or SPTFQMR should use the supplied FCVJTIMES, make the call

```
CALL FCVSPILSSETJAC(FLAG, IER)
```

with FLAG $\neq 0$ (see step 5 in §5.4 for details).

6. **Problem solution**

7. CVBBDPRE **Optional outputs**

   Optional outputs specific to the SPGMR, SPBCG, or SPTFQMR solver are listed in Table 5.2. To obtain the optional outputs associated with the CVBBDPRE module, make the following call:

   ```
   CALL FCVBBDOPT(LENRWBBD, LENIWBBD, NGEBBD)
   ```

   The arguments should be consistent with C type `long int`. Their returned values are as follows: LENRWBBD is the length of real preconditioner work space, in `realtype` words. LENIWBBD is the length of integer preconditioner work space, in integer words. These sizes are local to the current processor. NGEBBD is the number of $g(t, y)$ evaluations (calls to FCVLOCFN) so far.

8. **Problem reinitialization**

   If a sequence of problems of the same size is being solved using the same linear solver (SPGMR, SPBCG, or SPTFQMR) in combination with the CVBBDPRE preconditioner, then the CVODE package can be re-initialized for the second and subsequent problems by calling FCVREINIT, following which a call to FCVBBDINIT may or may not be needed. If the input arguments are the same, no FCVBBDINIT call is needed. If there is a change in input arguments other than MU or ML, then the user program should make the call

   ```
   CALL FCVBBDREINIT(NLOCAL, MUDQ, MLDQ, DQRELY, IER)
   ```

   This reinitializes the CVBBDPRE preconditioner, but without reallocating its memory. The arguments of the FCVBBDREINIT routine have the same names and meanings as those of FCVBBDINIT. If the value of MU or ML is being changed, then a call to FCVBBDINIT must be made. Finally, if there is a change in any of the linear solver inputs, then a call to FCVSPGMR, FCVSPBCG, or FCVSPTFQMR must also be made; in this case the linear solver memory is reallocated.

9. **Memory deallocation**

   (The memory allocated for the FCVBBD module is deallocated automatically by FCVFREE.)

10. **User-supplied routines**

    The following two routines must be supplied for use with the CVBBDPRE module:

    ```
    SUBROUTINE FCVGLOCFN (NLOC, T, YLOC, GLOC, IPAR, RPAR, IER)
    DIMENSION YLOC(*), GLOC(*), IPAR(*), RPAR(*)
    ```

    This routine is to evaluate the function $g(t, y)$ approximating f (possibly identical to f), in terms of T $= t$, and the array YLOC (of length NLOC), which is the sub-vector of y local to this processor. The resulting (local) sub-vector is to be stored in the array GLOC. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. IER is

an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if FCVGLOCFN failed unrecoverably (in which case the integration is halted).

```
SUBROUTINE FCVCOMMFN (NLOC, T, YLOC, IPAR, RPAR, IER)
DIMENSION YLOC(*), IPAR(*), RPAR(*)
```

This routine is to perform the inter-processor communication necessary for the FCVGLOCFN routine. Each call to FCVCOMMFN is preceded by a call to the right-hand side routine FCVFUN with the same arguments T and YLOC. The arrays IPAR (of integers) and RPAR (of reals) contain user data and are the same as those passed to FCVMALLOC. IER is an error return flag (currently not used; set IER=0). Thus FCVCOMMFN can omit any communications done by FCVFUN if relevant to the evaluation of GLOC. IER is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case CVODE will attempt to correct), or a negative value if FCVCOMMFN failed unrecoverably (in which case the integration is halted).

The subroutine FCVCOMMFN must be supplied even if it is not needed and must return IER=0.

Optionally, the user can supply a routine FCVJTIMES for the evaluation of Jacobian-vector products, as described above in step 5 in §5.4.

Chapter 6

Description of the NVECTOR module

The SUNDIALS solvers are written in a data-independent manner. They all operate on generic vectors (of type `N_Vector`) through a set of operations defined by the particular NVECTOR implementation. Users can provide their own specific implementation of the NVECTOR module, or use one of four provided within SUNDIALS – a serial implementation and three parallel implementations. The generic operations are described below. In the sections following, the implementations provided with SUNDIALS are described.

The generic `N_Vector` type is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the vector, and an *ops* field pointing to a structure with generic vector operations. The type `N_Vector` is defined as

```
typedef struct _generic_N_Vector *N_Vector;

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, long int *, long int *);
  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 6.2 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 6.1. It is recommended that a user-supplied NVECTOR implementation use the `SUNDIALS_NVEC_CUSTOM` identifier.

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

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

Table 6.2: 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 6.1.
`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 nonzero elements of the `N_Vector id`: $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i \mathrm{sign}(id_i))^2\right)/n}$.
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 `TRUE` if all components of `x` are nonzero (successful inversion) and returns `FALSE` 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 `FALSE` if any element failed the constraint test and assigned to `TRUE` 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.

6.1 The NVECTOR_SERIAL implementation

The serial implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_SERIAL, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, and a boolean flag *own_data* which specifies the ownership of *data*.

```
struct _N_VectorContent_Serial {
  long int length;
  booleantype own_data;
  realtype *data;
};
```

The header file to be included when using this module is `nvector_serial.h`.

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

- `NV_CONTENT_S`

  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 6.2. Their names are obtained from those in Table 6.2 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(long int vec_length);
  ```

- `N_VNewEmpty_Serial`

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

  ```
  N_Vector N_VNewEmpty_Serial(long int 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(long int 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.

  ```
  long int 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);
  ```

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* = FALSE. N_VDestroy_Serial and N_VDestroyVectorArray_Serial will not attempt to free the pointer *data* for any N_Vector with *own_data* set to FALSE. In such a case, it is the user's responsibility to deallocate the *data* pointer.

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

For solvers that include a Fortran interface module, the NVECTOR_SERIAL module also includes a Fortran-callable function FNVINITS(code, NEQ, IER), to initialize this NVECTOR_SERIAL module. Here code is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); NEQ is the problem size (declared so as to match C type long int); and IER is an error return flag equal 0 for success and -1 for failure.

6.2 The NVECTOR_PARALLEL implementation

The NVECTOR_PARALLEL implementation of the NVECTOR module provided with SUNDIALS is based on MPI. It defines the *content* field of N_Vector to be a structure containing the global and local lengths of the vector, a pointer to the beginning of a contiguous local data array, an MPI communicator, and a boolean flag *own_data* indicating ownership of the data array *data*.

```
struct _N_VectorContent_Parallel {
  long int local_length;
  long int global_length;
  booleantype own_data;
  realtype *data;
  MPI_Comm comm;
};
```

The header file to be included when using this module is nvector_parallel.h.

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

- NV_CONTENT_P

  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 6.2 Their names are obtained from those in Table 6.2 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,
                         long int local_length,
                         long int 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,
                              long int local_length,
                              long int 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,
                          long int local_length,
                          long int 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).

```
long int N_VGetLength_Parallel(N_Vector v);
```

- N_VGetLocalLength_Parallel

  This function returns the local vector length.

```
long int N_VGetLocalLength_Parallel(N_Vector v);
```

- N_VPrint_Parallel

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

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

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* = FALSE. N_VDestroy_Parallel and N_VDestroyVectorArray_Parallel will not attempt to free the pointer *data* for any N_Vector with *own_data* set to FALSE. In such a case, it is the user's responsibility to deallocate the *data* pointer.

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

For solvers that include a Fortran interface module, the NVECTOR_PARALLEL module also includes a Fortran-callable function `FNVINITP(COMM, code, NLOCAL, NGLOBAL, IER)`, to initialize this NVECTOR_PARALLEL module. Here `COMM` is the MPI communicator, `code` is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); `NLOCAL` and `NGLOBAL` are the local and global vector sizes, respectively (declared so as to match C type `long int`); and IER is an error return flag equal 0 for success and -1 for failure. NOTE: If the header file `sundials_config.h` defines `SUNDIALS_MPI_COMM_F2C` to be 1 (meaning the MPI implementation used to build SUNDIALS includes the `MPI_Comm_f2c` function), then `COMM` can be any valid MPI communicator. Otherwise, `MPI_COMM_WORLD` will be used, so just pass an integer value as a placeholder.

6.3 The NVECTOR_OPENMP implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR_OPENMP, and an implementation using Pthreads, called NVECTOR_PTHREADS. Testing has shown that vectors should be of length at least 100,000 before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The OpenMP NVECTOR implementation provided with SUNDIALS, NVECTOR_OPENMP, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag *own_data* which specifies the ownership of *data*, and the number of threads. Operations on the vector are threaded using OpenMP.

```
struct _N_VectorContent_OpenMP {
  long int length;
  booleantype own_data;
  realtype *data;
  int num_threads;
};
```

The header file to be included when using this module is `nvector_openmp.h`.

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

- `NV_CONTENT_OMP`

  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 6.2. Their names are obtained from those in Table 6.2 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(long int 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(long int 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(long int 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.

  ```
  long int N_VGetLength_OpenMP(N_Vector v);
  ```

- `N_VPrint_OpenMP`

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

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

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* = `FALSE`. `N_VDestroy_OpenMP` and `N_VDestroyVectorArray_OpenMP` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.

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

For solvers that include a Fortran interface module, the NVECTOR_OPENMP module also includes a Fortran-callable function `FNVINITOMP(code, NEQ, NUMTHREADS, IER)`, to initialize this module. Here `code` is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); NEQ is the problem size (declared so as to match C type `long int`); NUMTHREADS is the number of threads; and IER is an error return flag equal 0 for success and -1 for failure.

6.4 The NVECTOR_PTHREADS implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR_OPENMP, and an implementation using Pthreads, called NVECTOR_PTHREADS. Testing has shown that vectors should be of length at least 100,000 before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The Pthreads NVECTOR implementation provided with SUNDIALS, denoted NVECTOR_PTHREADS, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag *own_data* which specifies the ownership of *data*, and the number of threads. Operations on the vector are threaded using POSIX threads (Pthreads).

```
struct _N_VectorContent_Pthreads {
  long int length;
  booleantype own_data;
  realtype *data;
  int num_threads;
};
```

The header file to be included when using this module is `nvector_pthreads.h`.

The following six macros are provided to access the content of an NVECTOR_PTHREADS vector. The suffix `_PT` in the names denotes the Pthreads version.

- `NV_CONTENT_PT`

  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 6.2. Their names are obtained from those in Table 6.2 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(long int 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(long int 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(long int 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.

  ```
  long int 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);
  ```

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* = `FALSE`. `N_VDestroy_Pthreads` and `N_VDestroyVectorArray_Pthreads` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.

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

For solvers that include a Fortran interface module, the NVECTOR_PTHREADS module also includes a Fortran-callable function `FNVINITPTS(code, NEQ, NUMTHREADS, IER)`, to initialize this module. Here `code` is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); NEQ is the problem size (declared so as to match C type `long int`); NUMTHREADS is the number of threads; and IER is an error return flag equal 0 for success and -1 for failure.

6.5 The NVECTOR_PARHYP implementation

The NVECTOR_PARHYP implementation of the NVECTOR module provided with SUNDIALS is a wrapper around *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 {
  long int local_length;
  long int global_length;
  booleantype own_parvector;
  MPI_Comm comm;
  hypre_ParVector *x;
};
```

The header file to be included when using this module is `nvector_parhyp.h`. Unlike native SUNDIALS vector types, NVECTOR_PARHYP does not provide macros to access its member variables.

The NVECTOR_PARHYP module defines implementations of all vector operations listed in Table 6.2, except for `N_VSetArrayPointer` and `N_VGetArrayPointer`, because accessing raw vector data is

handled by low-level *hypre* functions. As such, this vector is not available for use with SUNDIALS Fortran interfaces. When access to raw vector data is needed, one should extract *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 [19] and the `ark_diurnal_kry_ph.c` example program for ARKODE [25].

The names of parhyp methods are obtained from those in Table 6.2 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,
                              long int local_length,
                              long int 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 content of a parhyp vector to stdout.

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

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 `FALSE`. `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 `FALSE`. In such a case, it is the user's responsibility to delete the underlying vector.

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

6.6 The NVECTOR_PETSC implementation

The NVECTOR_PETSC module is an NVECTOR wrapper around the PETSc vector. It defines the *content* field of a `N_Vector` to be a structure containing the global and local lengths of the vector, a pointer to the PETSc vector, an MPI communicator, and a boolean flag *own_data* indicating ownership of the wrapped PETSc vector.

```
struct _N_VectorContent_Petsc {
  long int local_length;
  long int global_length;
  booleantype own_data;
  Vec *pvec;
  MPI_Comm comm;
};
```

The header file to be included when using this module is `nvector_petsc.h`. 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 6.2, except for `N_VGetArrayPointer` and `N_VSetArrayPointer`. As such, this vector cannot be used with SUNDIALS Fortran interfaces. When access to raw vector data is needed, it is recommended to extract the PETSc vector first, and then use PETSc methods to access the data. Usage examples of NVECTOR_PETSC are provided in example programs for IDA [20].

The names of vector operations are obtained from those in Table 6.2 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,
                             long int local_length,
                             long int 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_VCloneEmptyVectorArray_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 content of a wrapped PETSc vector to stdout.

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

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 `FALSE`. `N_VDestroy_Petsc` and `N_VDestroyVectorArray_Petsc` will not attempt to free the pointer *pvec* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *pvec* pointer.

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

6.7 NVECTOR Examples

There are `NVector` examples that may be installed for each implementation: serial, parallel, OpenMP, and Pthreads. 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.

6.8 NVECTOR functions used by CVODE

In Table 6.3 below, we list the vector functions in the NVECTOR module used within the CVODE package. The table also shows, for each function, which of the code modules uses the function. The CVODE column shows function usage within the main integrator module, while the remaining seven columns show function usage within each of the eight CVODE linear solvers, the CVBANDPRE and CVBBDPRE preconditioner modules, and the FCVODE module. Here CVDLS stands for CVDENSE and CVBAND; CVSPILS stands for CVSPGMR, CVSPBCG, and CVSPTFQMR; and CVSLS stands for CVKLU and CVSUPERLUMT.

There is one subtlety in the CVSPILS column hidden by the table, explained here for the case of the CVSPGMR module. The `N_VDotProd` function is called both within the interface file `cvode_spgmr.c` and within the implementation files `sundials_spgmr.c` and `sundials_iterative.c` for the generic SPGMR solver upon which the CVSPGMR solver is built. Also, although `N_VDiv` and `N_VProd` are not called within the interface file `cvode_spgmr.c`, they are called within the implementation file `sundials_spgmr.c`, and so are required by the CVSPGMR solver module. Analogous statements apply to the CVSPBCG and CVSPTFQMR modules, except that they do not use `sundials_iterative.c`. This issue does not arise for the other three CVODE linear solvers because the generic DENSE and BAND solvers (used in the implementation of CVDENSE and CVBAND) do not make calls to any vector functions and CVDIAG is not implemented using a generic diagonal solver.

At this point, we should emphasize that the CVODE user does not need to know anything about the usage of vector functions by the CVODE code modules in order to use CVODE. The information is presented as an implementation detail for the interested reader.

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

Table 6.3: List of vector functions usage by CVODE code modules

	CVODE	CVDLS	CVDIAG	CVSPILS	CVSLS	CVBANDPRE	CVBBDPRE	FCVODE
N_VGetVectorID								
N_VClone	✓		✓	✓				
N_VCloneEmpty								✓
N_VDestroy	✓		✓	✓				
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_VCompare			✓					
N_VInvTest			✓					

Chapter 7

Description of the SUNMatrix module

For problems that involve direct methods for solving linear systems, the SUNDIALS solvers not only operate on generic vectors, but they also operate 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, for all of the direct linear solvers provided within SUNDIALS, we provide four SUNMATRIX implementations: dense, banded, diagonal and sparse. The generic operations are described below, and descriptions of the implementations provided with SUNDIALS follow.

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

```
typedef struct _generic_SUNMatrix *SUNMatrix;

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

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

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

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

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

Matrix ID	Matrix type	ID Value
SUNMATRIX_DENSE	Dense $m \times n$ matrix	0
SUNMATRIX_BAND	Band $m \times m$ matrix	1
SUNMATRIX_DIAGONAL	Diagonal $m \times m$ matrix	2
SUNMATRIX_SPARSE	Sparse (CSR or CSC) $m \times n$ matrix	3
SUNMATRIX_CUSTOM	User-provided custom matrix	4

illustrate this point we show below the implementation of a typical matrix operation from the generic SUNMATRIX module, namely `SUNMatZero`, which sets all values of a matrix `A` to zero, returning a flag denoting a successful/failed operation:

```
int SUNMatZero(SUNMatrix A)
{
  return((int) A->ops->zero(A));
}
```

Table 7.2 contains a complete list of all matrix operations defined by the generic SUNMATRIX module.

A particular implementation of the SUNMATRIX module must:

- Specify the *content* field of the `SUNMatrix` object.
- Define and implement a minimal subset of the matrix operations. See the documentation for each SUNDIALS solver to determine which SUNMATRIX operations they require.

  Note that the names of these routines should be unique to that implementation in order to permit using more than one SUNMATRIX module (each with different `SUNMatrix` internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free a `SUNMatrix` with the new *content* field and with *ops* pointing to the new matrix operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined `SUNMatrix` (e.g., a routine to print the content for debugging purposes).
- Optionally, provide accessor macros or functions as needed for that particular implementation to be used to access different parts in the *content* field of the newly defined `SUNMatrix`.

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

Table 7.2: Description of the `SUNMatrix` operations

Name	Usage and Description
`SUNMatGetID`	`id = SUNMatGetID(A);` Returns the type identifier for the matrix `A`. It is used to determine the matrix implementation type (e.g. band, dense, 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 7.1.

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.
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.

7.1 The SUNMatrix_Dense implementation

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

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

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

M - number of rows

N - number of columns

data - pointer to a contiguous block of `realtype` variables. The elements of the dense matrix are stored columnwise, i.e. the (`i`,`j`)-th element of a dense SUNMATRIX `A` (with $0 \le$ `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 be included when using this module is sunmatrix/sunmatrix_dense.h.
The following eight 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 routine 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 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.

  Similarlly, 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 gives access to the individual columns and entries of the data array of a dense SUNMatrix.

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

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

  Implementation:

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

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

- `SUNDenseMatrix`

  This 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 $M-1$.

  ```
  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 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 so as to match C type `long int`); and `ier` is an error return flag equal 0 for success and -1 for failure (declared so as to match C type `int`). Additionally, when using ARKODE with non-identity mass matrix, the Fortran-callable function `FSUNDenseMassMatInit(M, N, ier)` initializes this SUNMATRIX_DENSE module for storing the mass matrix.

7.2 The SUNMatrix_Band implementation

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

```
struct _SUNMatrixContent_Band {
  sunindextype M;
  sunindextype N;
  sunindextype mu;
  sunindextype ml;
  sunindextype s_mu;
  sunindextype ldim;
  realtype *data;
  sunindextype ldata;
  realtype **cols;
};
```

A diagram of the underlying data representation in a banded matrix is shown in Figure 10.1. 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` < min(`M`,`N`)

ml - lower half-bandwidth, 0 ≤ `ml` < min(`M`,`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.

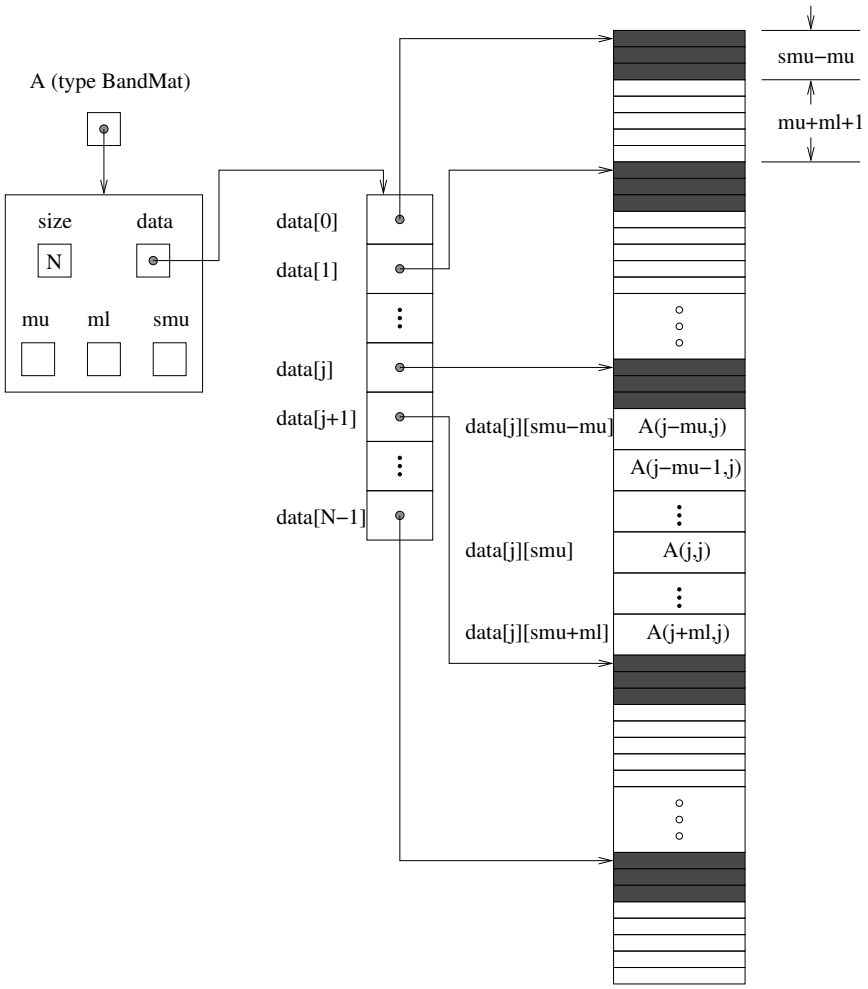


Figure 7.1: Diagram of the storage for the SUNMATRIX_BAND module. Here `A` is an $N \times N$ band matrix with upper and lower half-bandwidths `mu` and `ml`, respectively. The rows and columns of `A` are numbered from 0 to $N-1$ and the (i,j)-th element of `A` is denoted `A(i,j)`. The greyed out areas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

ldim - leading dimension (`ldim` $\geq$ `s_mu`)

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`·(`s_mu`+`ml`+1)

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, $j-$`mu` $\leq i \leq j+$`ml`.

The header file to be included when using this module is `sunmatrix/sunmatrix_band.h`.

The following thirteen macros are provided to access the content of a SUNMATRIX_BAND matrix. The prefix SM_ in the names denotes that these macros are for *SUNMatrix* implementations, and the suffix _B denotes that these are specific to the *banded* version.

- SM_CONTENT_B

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

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

  Implementation:

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

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

  These macros give individual access 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.

  Similarlly, 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 gives access to the individual columns and entries of the data array of a banded SUNMatrix.

  The assignments SM_ELEMENT_B(A,i,j) = a_ij and a_ij = SM_ELEMENT_B(A,i,j) reference the (i,j)-th element of the $N \times N$ band matrix A, where $0 \le$ 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 $N \times N$ band matrix A, $0 \le$ j $\le N-1$. The type of the expression SM_COLUMN_B(A,j) is realtype *. The pointer returned by the call SM_COLUMN_B(A,j) can be treated as an array which is indexed from $-$mu to ml.

  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 $\texttt{j}-\texttt{mu} \leq \texttt{i} \leq \texttt{j}+\texttt{ml}$.

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

- `SUNBandMatrix`

  This 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`, if the matrix will be used by the SUNLINSOL_BAND module then `smu` should be at least min(`N`-1,`mu`+`ml`); otherwise `smu` should be at least `mu`.

  ```
  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 jth 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 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 so as to match C type `long int`); and `ier` is an error return flag equal 0 for success and -1 for failure (declared so as to match C type `int`). Additionally, when using ARKODE with non-identity mass matrix, the Fortran-callable function `FSUNBandMassMatInit(N, mu, ml, smu, ier)` initializes this SUNMATRIX_BAND module for storing the mass matrix.

7.3 The SUNMatrix_Diagonal implementation

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

```
struct _SUNMatrixContent_Diagonal {
  N_Vector d;
};
```

The entry of the *content* field contain the following information:

d - generic NVECTOR object containing the diagonal of the SUNMATRIX.

The header file to be included when using this module is `sunmatrix/sunmatrix_diagonal.h`.
The following two macros are provided to access the content of a SUNMATRIX_DIAGONAL matrix. The prefix `SM_` in the names denotes that these macros are for *SUNMatrix* implementations, and the suffix `_DIAG` denotes that these are specific to the *diagonal* version.

- `SM_CONTENT_DIAG`

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

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

  Implementation:

  ```
  #define SM_CONTENT_DIAG(A)  ( (SUNMatrixContent_Diagonal)(A->content) )
  ```

- `SM_DATA_DIAG`

  This macro gives access to the NVECTOR `d` that defines the diagonal matrix.

  The assignment `A_data = SM_DATA_DIAG(A)` sets `A_data` to be the NVECTOR storing the matrix diagonal. The assignment `SM_DATA_DIAG(A) = A_data` sets the NVECTOR matrix diagonal to `A_data`.

  Implementation:

  ```
  #define SM_DATA_DIAG(A)     ( SM_CONTENT_DIAG(A)->d )
  ```

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

- `SUNDiagonalMatrix`

  This function creates and allocates memory for a diagonal `SUNMatrix`. Its argument is a template NVECTOR object.

  ```
  SUNMatrix SUNDiagonalMatrix(N_Vector tmpl);
  ```

- `SUNDiagonalMatrix_Diag`

  This function returns the NVECTOR containing the diagonal of the `SUNMatrix`.

  ```
  N_Vector SUNDiagonalMatrix_Diag(SUNMatrix A);
  ```

Notes

- To access the components of a diagonal `SUNMatrix A`, you must first retrieve the NVECTOR containing the matrix diagonal via `A_data = SM_DATA_DIAG(A)` or `A_data = SUNDiagonalMatrix_Diag(A)`, and then access the entries of the NVECTOR using whichever macros or functions are appropriate for that NVECTOR implementation.

- Within the `SUNMatMatvec_Diagonal` routine, internal consistency checks are performed to ensure that the matrix is called with consistent NVECTOR implementations. This consistency check merely ensures that the matrix diagonal, as well as the input NVECTOR objects `x` and `y` all have identical `N_Vector_ID`. As such, this routine may be used with any SUNDIALS-supplied or user-supplied NVECTOR implementation.

For solvers that include a Fortran interface module, the SUNMATRIX_DIAGONAL module also includes the Fortran-callable function `FSUNDiagonalMatInit(code, ier)` to initialize this SUNMATRIX_DIAGONAL module for a given SUNDIALS solver. Here `code` is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); and `ier` is an error return flag equal 0 for success and -1 for failure (declared so as to match C type `int`). This routine must be called *after* a corresponding NVECTOR has been initialized by a call to `FNVInit*`. Additionally, when using ARKODE with non-identity mass matrix, the Fortran-callable function `FSUNDiagonalMassMatInit(ier)` initializes this SUNMATRIX_DIAGONAL module for storing the mass matrix.

7.4 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 10.2 (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 `rowvals` 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};
rowvals = &indexvals;
colptrs = &indexptrs;
colvals = NULL;
rowptrs = NULL;
```

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. The work associated with operations on the sparse matrix is proportional to this value and so one should use the best understanding of the number of nonzeroes here.
The header file to be included when using this module is `sunmatrix/sunmatrix_sparse.h`.

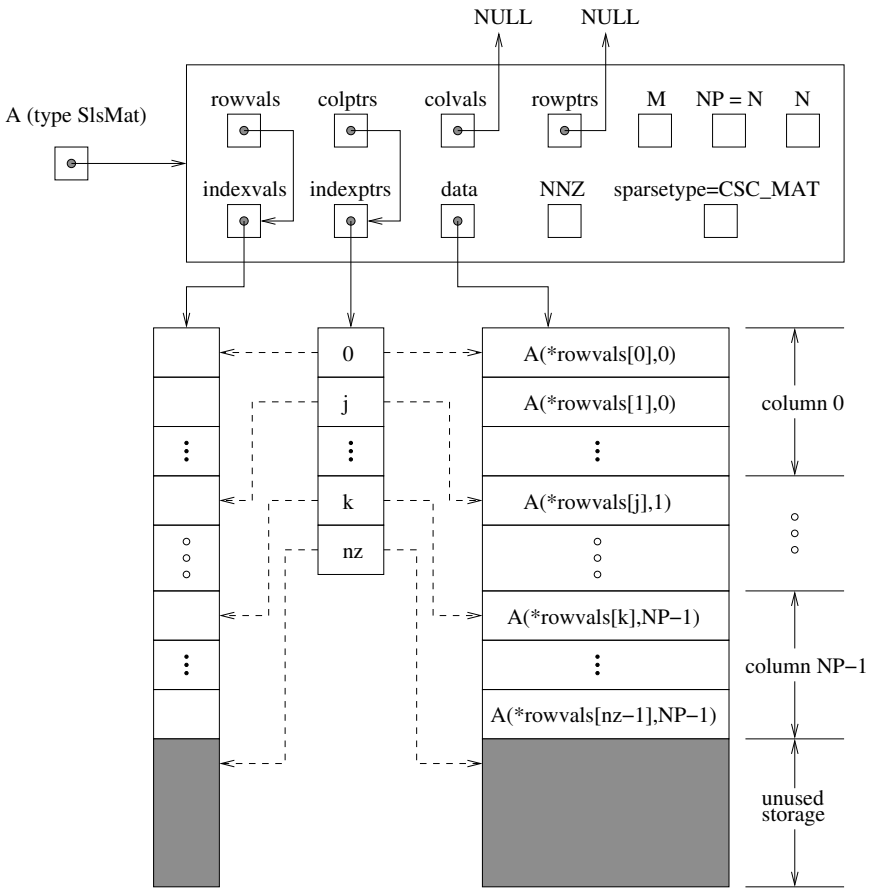


Figure 7.2: Diagram of the storage for a compressed-sparse-column matrix. Here `A` is an $M \times N$ sparse matrix with storage for up to `NNZ` nonzero entries (the allocated length of both `data` and `indexvals`). The entries in `indexvals` may assume values from 0 to $M-1$, corresponding to the row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

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

- SM_CONTENT_S

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

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

  Implementation:

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

- SM_ROWS_S, SM_COLUMNS_S, SM_NNZ_S, SM_NP_S, and SM_SPARSETYPE_S

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

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

  Implementation:

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

- SM_DATA_S, SM_INDEXVALS_S, and SM_INDEXPTRS_S

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

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

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

  Implementation:

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

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

- SUNSparseMatrix

  This function creates and allocates memory for a sparse SUNMatrix. Its arguments are the number of rows and columns of the matrix, M and N, the maximum number of nonzeros to be stored in the matrix, NNZ, and a flag indicating whether to use CSR or CSC format, 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 number of allocated space for nonzeros 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_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 allocated nonzero storage for the sparse 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 this is the location of the first entry of each row in the data and indexvalues arrays, for CSC 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 so as 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 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 non-identity mass matrix, the Fortran-callable function `FSUNSparseMassMatInit(M, N, NNZ, sparsetype, ier)` initializes this SUNMATRIX_SPARSE module for storing the mass matrix.

7.5 SUNMatrix Examples

There are `SUNMatrix` examples that may be installed for each implementation: band, dense, diagonal 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 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, 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 has different behavior depending on whether A is square. If it is square, it clones and copies A to a new matrix B, computes $B = 3B + I$ using `SUNMatScaleAddI`, clones y to new vectors w and z, computes $z = Bx$ using `SUNMatMatvec`, computes $w = 3y + x$ using `N_VLinearSum`, and verifies that $w == z$. If A is not square, it just clones y to a new vector z, computes $z = Ax$ using `SUNMatMatvec`, and verifies that $y == z$.
- `Test_SUNMatSpace` verifies that `SUNMatSpace` can be called, and outputs the results to `stdout`.

7.6 SUNMatrix functions used by CVODE

In Table 7.3 below, we list the matrix functions in the SUNMATRIX module used within the CVODE package. The table also shows, for each function, which of the code modules uses the function. Neither the main CVODE 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 CVODE user does not need to know anything about the usage of matrix functions by the CVODE code modules in order to use CVODE. The information is presented as an implementation detail for the interested reader.

The matrix functions listed in Table 7.2 that are *not* used by CVODE are: `SUNMatScaleAdd` and `SUNMatMatvec`. Therefore a user-supplied SUNMATRIX module for CVODE could omit these functions.

Table 7.3: List of matrix functions usage by CVODE code modules

	CVDLS	CVBANDPRE	CVBBDPRE
SUNMatGetID	✓		
SUNMatClone	✓		
SUNMatDestroy	✓	✓	✓
SUNMatZero	✓	✓	✓
SUNMatCopy	✓	✓	✓
SUNMatScaleAddI	✓	✓	✓
SUNMatSpace	✓	✓	✓

Chapter 8

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 of eight SUNLINSOL modules, as well as four 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 be 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

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

where

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

and where

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

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*,
                                    ATSetupFn, 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 8.2 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

Table 8.1: 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

such, for each routine we specify its intended use. If a custom SUNLINSOL module is to be 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:

- 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 be used 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 8.1. 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 `SUNLINEARSOLVER_DIRECT` SUNLINSOL objects, "Spils" interfaces require `SUNLINEARSOLVER_ITERATIVE` objects, and "Cls" interfaces require `SUNLINEARSOLVER_CUSTOM` objects.

Table 8.2: 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 8.1.

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 8.3.
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 8.3.
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 8.3. **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, ATSetup, ATimes);` (Iterative/Custom linear solvers only) Provides `ATSetupFn` and `ATimesFn` function pointers, as well as a `void *` pointer to a data structure used by these routines, to a linear solver object. This function will be called by a SUNDIALS solver, who will either provide this via difference-quotients and vector operations, or by calling integrator-specific, 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 8.3.

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 (8.1)-(8.2). This routine will be called by a SUNDIALS solver, who will provide translation between the generic `Pset` and `Psol` calls and the integrator-specific, 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 8.3.
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 (8.1)-(8.2). Similarly, `s2` is an NVECTOR containing the diagonal of S_2 from equations (8.1)-(8.2). 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 8.3.
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.

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.

Table 8.3: 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_ASET_FAIL_UNREC	-4	an unrecoverable failure occurred in the `ATSetup` routine
SUNLS_ATIMES_FAIL_UNREC	-5	an unrecoverable failure occurred in the `ATimes` routine
SUNLS_PSET_FAIL_UNREC	-6	an unrecoverable failure occurred in the `Pset` routine
SUNLS_PSOLVE_FAIL_UNREC	-7	an unrecoverable failure occurred in the `Psolve` routine
SUNLS_PACKAGE_FAIL_UNREC	-8	an unrecoverable failure occurred in an external linear solver package
SUNLS_GS_FAIL	-9	a failure occurred during Gram-Schmidt orthogonalization (SUNLINSOL_SPGMR/SUNLINSOL_SPFGMR)
SUNLS_QRSOL_FAIL	-10	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

Name	Value	Description
SUNLS_CONV_FAIL	2	an iterative solver did not converge (and the residual was not reduced)
SUNLS_ASET_FAIL_REC	3	a recoverable failure occurred in the ATSetup routine
SUNLS_ATIMES_FAIL_REC	4	a recoverable failure occurred in the ATimes routine
SUNLS_PSET_FAIL_REC	5	a recoverable failure occurred in the Pset routine
SUNLS_PSOLVE_FAIL_REC	6	a recoverable failure occurred in the Psolve routine
SUNLS_PACKAGE_FAIL_REC	7	a recoverable failure occurred in an external linear solver package
SUNLS_QRFACT_FAIL	8	a singular matrix was encountered during a QR factorization (SUNLINSOL_SPGMR/SUNLINSOL_SPFGMR)
SUNLS_LUFACT_FAIL	9	a singular matrix was encountered during a (SUNLINSOL_DENSE/SUNLINSOL_BAND/SUNLINSOL_DIAGONAL)

8.1 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:

- In the "setup" call, this 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.
- In the "solve" call, this performs pivoting, 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 be included when using this module is sunlinsol/sunlinsol_dense.h.

The SUNLINSOL_DENSE module defines dense implementations of all "direct" linear solver operations listed in Table 8.2:

- SUNLinSolGetType_Dense

- `SUNLinSolInitialize_Dense` – this does nothing, since all consistency checks were 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 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 input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); `ier` is an error return flag equal 0 for success and -1 for failure (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 non-identity mass matrix, the Fortran-callable function `FSUNMassDenseLinSolInit(ier)` initializes this SUNLINSOL_DENSE module for solving mass matrix linear systems.

8.2 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:

- In the "setup" call, this 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.
- In the "solve" call, this performs pivoting, 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 be included when using this module is `sunlinsol/sunlinsol_band.h`.

The SUNLINSOL_BAND module defines band implementations of all "direct" linear solver operations listed in Table 8.2:

- `SUNLinSolGetType_Band`
- `SUNLinSolInitialize_Band` – this does nothing, since all consistency checks were 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 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 input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); `ier` is an error return flag equal 0 for success and -1 for failure (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 non-identity mass matrix, the Fortran-callable function `FSUNMassBandLinSolInit(ier)` initializes this SUNLINSOL_BAND module for solving mass matrix linear systems.

8.3 The SUNLinearSolver_Diagonal implementation

The diagonal implementation of the SUNLINSOL module provided with SUNDIALS, SUNLINSOL_DIAGONAL, is designed to be used with the corresponding SUNMATRIX_DIAGONAL matrix type (and any of the NVECTOR implementations). The SUNLINSOL_DIAGONAL module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_Diagonal {
  long int last_flag;
};
```

The entry of the *content* field consists of only a long integer containing the last error return flag from SUNLINSOL_DIAGONAL functions.

This solver is constructed to perform the following operations:

- In the "setup" call, this inverts a diagonal matrix using the NVECTOR routine `N_VInvTest`. This inverse is stored in-place on the input SUNMATRIX_DIAGONAL object A.
- In the "solve" call, this performs the diagonal solve using the matrix inverse by calling the NVECTOR routine `N_VProd`.

The header file to be included when using this module is `sunlinsol/sunlinsol_diagonal.h`.

The SUNLINSOL_DIAGONAL module defines diagonal implementations of all "direct" linear solver operations listed in Table 8.2:

- `SUNLinSolGetType_Diagonal`
- `SUNLinSolInitialize_Diagonal` – this does nothing, since all consistency checks were performed at solver creation.
- `SUNLinSolSetup_Diagonal` – this inverts the input diagonal matrix.
- `SUNLinSolSolve_Diagonal` – this multiplies with the input diagonal matrix.
- `SUNLinSolLastFlag_Diagonal`
- `SUNLinSolSpace_Diagonal` – this only returns information for the storage *within* the solver object, i.e. storage for `last_flag`.
- `SUNLinSolFree_Diagonal`

The module SUNLINSOL_DIAGONAL provides the following additional user-callable routine:

- `SUNDiagonalLinearSolver`

  This function creates and allocates memory for a diagonal `SUNLinearSolver`. Its arguments are an NVECTOR and SUNMATRIX, that it uses 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. Since a SUNMATRIX_DIAGONAL merely contains a NVECTOR, then this checks that both the input NVECTOR and the SUNMATRIX_DIAGONAL NVECTOR have the same vector type. If these do not match, then this routine will return `NULL`.

  Additionally, this routine checks that both of the requisite NVECTOR operations, `N_VInvTest` and `N_VProd`, are supplied by the NVECTOR implementation. If either are missing, then this routine will return `NULL`.

  ```
  SUNLinearSolver SUNDiagonalLinearSolver(N_Vector y, SUNMatrix A);
  ```

For solvers that include a Fortran interface module, the SUNLINSOL_DIAGONAL module also includes the Fortran-callable function `FSUNDiagLinSolInit(code, ier)` to initialize this SUNLINSOL_DIAGONAL module for a given SUNDIALS solver. Here `code` is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); `ier` is an error return flag equal 0 for success and -1 for failure (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 non-identity mass matrix, the Fortran-callable function `FSUNMassDiagLinSolInit(ier)` initializes this SUNLINSOL_DIAGONAL module for solving mass matrix linear systems.

8.4 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). Therfore 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 128-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:

- In the "setup" call, this 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.
- In the "solve" call, this performs pivoting, 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 be included when using this module is `sunlinsol/sunlinsol_lapackdense.h`.

The SUNLINSOL_LAPACKDENSE module defines dense implementations of all "direct" linear solver operations listed in Table 8.2:

- `SUNLinSolGetType_LapackDense`

- SUNLinSolInitialize_LapackDense – this does nothing, since all consistency checks were 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 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 input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); ier is an error return flag equal 0 for success and -1 for failure (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 non-identity mass matrix, the Fortran-callable function FSUNMassLapackDenseInit(ier) initializes this SUNLINSOL_LAPACKDENSE module for solving mass matrix linear systems.

8.5 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). Therfore 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 128-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:

- In the "setup" call, this 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.
- In the "solve" call, this performs pivoting, 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 be included when using this module is `sunlinsol/sunlinsol_lapackband.h`.

The SUNLINSOL_LAPACKBAND module defines band implementations of all "direct" linear solver operations listed in Table 8.2:

- `SUNLinSolGetType_LapackBand`
- `SUNLinSolInitialize_LapackBand` – this does nothing, since all consistency checks were 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 input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); `ier` is an error return flag equal 0 for success and -1 for failure (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 non-identity mass matrix, the Fortran-callable function `FSUNMassLapackBandInit(ier)` initializes this SUNLINSOL_LAPACKBAND module for solving mass matrix linear systems.

8.6 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]. Therfore 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 64-bit and 128-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 therfore constructed to perform the following operations:

- The first time that the "setup" routine is called, this performs the symbolic factorization, followed by an initial numerical factorization.
- On subsequent calls to the "setup" routine, this calls the appropriate KLU "refactor" routine, followed by estimates of the numerical conditioning using the relevant "rcond", and if necessary "condest", routine(s). If these estimates of the condition number are larger than $\varepsilon^{-2/3}$ (where ε is the double-precision unit roundoff), then a new factorization is performed.
- The module includes the routine `SUNKLUReInit`, that can be called by the user to force a full refactorization at the next "setup" call.
- In the "solve" call, this performs pivoting, forward and backward substitution using the stored `pivots` array and the sparse LU factors.

The header file to be included when using this module is `sunlinsol/sunlinsol_klu.h`.

The SUNLINSOL_KLU module defines dense implementations of all "direct" linear solver operations listed in Table 8.2:

- `SUNLinSolGetType_KLU`
- `SUNLinSolInitialize_KLU` – this sets the `first_factorize` flag to 1, forcing bot 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 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:

  1 The Jacobian matrix will be destroyed and a new one will be allocated based on the `nnz` value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup.

  2 Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of `nnz` given in the sparse matrix provided to the original constructor routine (or the previous `SUNKLUReInit` call).

  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 input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKODE); `ier` is an error return flag equal 0 for success and -1 for failure (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 non-identity mass matrix, the Fortran-callable function `FSUNMassKLUInit(ier)` initializes this SUNLINSOL_KLU module for solving mass matrix linear systems.

Additionally, 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`
- `FSUNMassKLUReInit(NNZ, reinit_type, ier)`
- `FSUNKLUSetOrdering(code, ordering, ier)` – `ordering` should be commensurate with a C `int`
- `FSUNMassKLUSetOrdering(ordering, ier)`

8.7 The SUNLinearSolver_SuperLUMT implementation

8.8 The SUNLinearSolver_SPGMR implementation

8.9 The SUNLinearSolver_SPFGMR implementation

8.10 The SUNLinearSolver_SPBCGS implementation

8.11 The SUNLinearSolver_SPTFQMR implementation

8.12 The SUNLinearSolver_PCG implementation

8.13 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.

8.14 SUNLinearSolver functions used by CVODE

In Table 8.4 below, we list the linear solver functions in the SUNLINSOL module used within the CVODE package. The table also shows, for each function, which of the code modules uses the function. In general, the main CVODE integrator considers three categories of linear solvers, *direct*, *iterative* and *custom*, with interfaces accessible in the CVODE header files `cvode_direct.h` (CVDLS), `cvode_spils.h` (CVSPILS) and `cvode_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 CVODE user does not need to know detailed usage of linear solver functions by the CVODE code modules in order to use CVODE. The information is presented as an implementation detail for the interested reader.

The linear solver functions listed in Table 8.2 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 CVODE does not call the `SUNLinSolLastFlag` directly, this routine is available for users to query linear solver issues directly.

Table 8.4: List of linear solver functions usage by CVODE code modules

	CVDLS	CVSPILS	CVCLS
SUNLinSolGetType	✓	✓	†
SUNLinSolSetATimes		✓	†
SUNLinSolSetPreconditioner		✓	†
SUNLinSolSetScalingVectors		✓	†
SUNLinSolInitialize	✓	✓	✓
SUNLinSolSetup	✓	✓	✓
SUNLinSolSolve	✓	✓	✓
SUNLinSolNumIters		✓	†
SUNLinSolResNorm		✓	†
SUNLinSolResid		✓	†
SUNLinSolLastFlag			
SUNLinSolFree	✓	✓	✓
SUNLinSolSpace	✓	✓	†

Chapter 9

Providing Alternate Linear Solver Modules

The central CVODE module interfaces with a linear solver module by way of calls to four functions. These are denoted here by `linit`, `lsetup`, `lsolve`, and `lfree`. Briefly, their purposes are as follows:

- `linit`: initialize memory specific to the linear solver;
- `lsetup`: evaluate and preprocess the Jacobian or preconditioner;
- `lsolve`: solve the linear system;
- `lfree`: free the linear solver memory.

A linear solver module must also provide a user-callable **specification function** (like those described in §4.5.3) which will attach the above four functions to the main CVODE memory block. The CVODE memory block is a structure defined in the header file `cvode_impl.h`. A pointer to such a structure is defined as the type `CVodeMem`. The four fields in a `CvodeMem` structure that must point to the linear solver's functions are `cv_linit`, `cv_lsetup`, `cv_lsolve`, and `cv_lfree`, respectively. Note that of the four interface functions, only the `lsolve` function is required. The `lfree` function must be provided only if the solver specification function makes any memory allocation. For any of the functions that are *not* provided, the corresponding field should be set to `NULL`. The linear solver specification function must also set the value of the field `cv_setupNonNull` in the CVODE memory block — to `TRUE` if `lsetup` is used, or `FALSE` otherwise.

Typically, the linear solver will require a block of memory specific to the solver, and a principal function of the specification function is to allocate that memory block, and initialize it. Then the field `cv_lmem` in the CVODE memory block is available to attach a pointer to that linear solver memory. This block can then be used to facilitate the exchange of data between the four interface functions.

If the linear solver involves adjustable parameters, the specification function should set the default values of those. User-callable functions may be defined that could, optionally, override the default parameter values.

We encourage the use of performance counters in connection with the various operations involved with the linear solver. Such counters would be members of the linear solver memory block, would be initialized in the `linit` function, and would be incremented by the `lsetup` and `lsolve` functions. Then, user-callable functions would be needed to obtain the values of these counters.

For consistency with the existing CVODE linear solver modules, we recommend that the return value of the specification function be 0 for a successful return, and a negative value if an error occurs. Possible error conditions include: the pointer to the main CVODE memory block is `NULL`, an input is illegal, the NVECTOR implementation is not compatible, or a memory allocation fails.

These four functions, which interface between CVODE and the linear solver module, necessarily have fixed call sequences. Thus, a user wishing to implement another linear solver within the CVODE

package must adhere to this set of interfaces. The following is a complete description of the call list for each of these functions. Note that the call list of each function includes a pointer to the main CVODE memory block, by which the function can access various data related to the CVODE solution. The contents of this memory block are given in the file `cvode_impl.h` (but not reproduced here, for the sake of space).

9.1 Initialization function

The type definition of `linit` is

`linit`

Definition
```
int (*linit)(CVodeMem cv_mem);
```

Purpose The purpose of `linit` is to complete initializations for the specific linear solver, such as counters and statistics. It should also set pointers to data blocks that will later be passed to functions associated with the linear solver. The `linit` function is called once only, at the start of the problem, during the first call to `CVode`.

Arguments `cv_mem` is the CVODE memory pointer of type `CVodeMem`.

Return value An `linit` function should return 0 if it has successfully initialized the CVODE linear solver, and a negative value otherwise.

9.2 Setup function

The type definition of `lsetup` is

`lsetup`

Definition
```
int (*lsetup)(CVodeMem cv_mem, int convfail, N_Vector ypred,
              N_Vector fpred, booleantype *jcurPtr,
              N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3);
```

Purpose The job of `lsetup` is to prepare the linear solver for subsequent calls to `lsolve`, in the solution of systems $Mx = b$, where M is some approximation to the Newton matrix, $I - \gamma\, \partial f/\partial y$. (See Eq.(2.5)). Here γ is available as `cv_mem->cv_gamma`.

The `lsetup` function may call a user-supplied function, or a function within the linear solver module, to compute needed data related to the Jacobian matrix $\partial f/\partial y$. Alternatively, it may choose to retrieve and use stored values of this data.

In either case, `lsetup` may also preprocess that data as needed for `lsolve`, which may involve calling a generic function (such as for LU factorization). This data may be intended either for direct use (in a direct linear solver) or for use in a preconditioner (in a preconditioned iterative linear solver).

The `lsetup` function is not called at every time step, but only as frequently as the solver determines that it is appropriate to perform the setup task. In this way, Jacobian-related data generated by `lsetup` is expected to be used over a number of time steps.

Arguments `cv_mem` is the CVODE memory pointer of type `CVodeMem`.

`convfail` is an input flag used to indicate any problem that occurred during the solution of the nonlinear equation on the current time step for which the linear solver is being used. This flag can be used to help decide whether the Jacobian data kept by a CVODE linear solver needs to be updated or not. Its possible values are:

- CV_NO_FAILURES: this value is passed to lsetup if either this is the first call for this step, or the local error test failed on the previous attempt at this step (but the Newton iteration converged).
- CV_FAIL_BAD_J: this value is passed to lsetup if (a) the previous Newton corrector iteration did not converge and the linear solver's setup function indicated that its Jacobian-related data is not current, or (b) during the previous Newton corrector iteration, the linear solver's solve function failed in a recoverable manner and the linear solver's setup function indicated that its Jacobian-related data is not current.
- CV_FAIL_OTHER: this value is passed to lsetup if during the current internal step try, the previous Newton iteration failed to converge even though the linear solver was using current Jacobian-related data.

ypred is the predicted y vector for the current CVODE internal step.

fpred is the value of the right-hand side at ypred, $f(t_n$, ypred$)$.

jcurPtr is a pointer to a boolean to be filled in by lsetup. The function should set *jcurPtr = TRUE if its Jacobian data is current after the call, and should set *jcurPtr = FALSE if its Jacobian data is not current. If lsetup calls for re-evaluation of Jacobian data (based on convfail and CVODE state data), it should return *jcurPtr = TRUE unconditionally; otherwise an infinite loop can result.

vtemp1

vtemp2

vtemp3 are temporary variables of type N_Vector provided for use by lsetup.

Return value The lsetup function should return 0 if successful, a positive value for a recoverable error, and a negative value for an unrecoverable error. On a recoverable error return, the solver will attempt to recover by reducing the step size.

9.3 Solve function

The type definition of lsolve is

lsolve

Definition
```
int (*lsolve)(CVodeMem cv_mem, N_Vector b, N_Vector weight,
              N_Vector ycur, N_Vector fcur);
```

Purpose The function lsolve must solve the linear system $Mx = b$, where M is some approximation to the Newton matrix, $I - \gamma J$, where $J = (\partial f/\partial y)(t_n, y_{cur})$ (see Eq.(2.5)), and the right-hand side vector, b, is input. Here γ is available as cv_mem->cv_gamma.

lsolve is called once per Newton iteration, hence possibly several times per time step.

If there is an lsetup function, this lsolve function should make use of any Jacobian data that was computed and preprocessed by lsetup, either for direct use, or for use in a preconditioner.

Arguments cv_mem is the CVODE memory pointer of type CVodeMem.

b is the right-hand side vector b. The solution is to be returned in the vector b.

weight is a vector that contains the error weights. These are the W_i of Eq.(2.6). This weight vector is included here to enable the computation of weighted norms needed to test for the convergence of iterative methods (if any) within the linear solver.

ycur is a vector that contains the solver's current approximation to $y(t_n)$.

fcur is a vector that contains the current right-hand side, $f(t_n$, ycur$)$.

Return value The `lsolve` function should return a positive value for a recoverable error and a negative value for an unrecoverable error. Success is indicated by a 0 return value. On a recoverable error return, the solver will attempt to recover, such as by calling the `lsetup` function with current arguments.

9.4 Memory deallocation function

The type definition of `lfree` is

`lfree`

Definition `int (*lfree)(CVodeMem cv_mem);`

Purpose The function `lfree` should free up any memory allocated by the linear solver.

Arguments The argument `cv_mem` is the CVODE memory pointer of type `CVodeMem`.

Return value The `lfree` function should return 0 if successful, or a nonzero if not.

Notes This function is called once a problem has been completed and the linear solver is no longer needed.

Chapter 10

General Use Linear Solver Components in SUNDIALS

In this chapter, we describe linear solver code components that are included in SUNDIALS, but which are of potential use as generic packages in themselves, either in conjunction with the use of SUNDIALS or separately.

These generic modules in SUNDIALS are organized in three families, the *dls* family, which includes direct linear solvers appropriate for sequential computations; the *sls* family, which includes sparse matrix solvers; and the *spils* family, which includes scaled preconditioned iterative (Krylov) linear solvers. The solvers in each family share common data structures and functions.

The *dls* family contains the following two generic linear solvers:

- The DENSE package, a linear solver for dense matrices either specified through a matrix type (defined below) or as simple arrays.
- The BAND package, a linear solver for banded matrices either specified through a matrix type (defined below) or as simple arrays.

Note that this family also includes the Blas/Lapack linear solvers (dense and band) available to the SUNDIALS solvers, but these are not discussed here.

The *sls* family contains a sparse matrix package and interfaces between it and two sparse direct solver packages:

- The KLU package, a linear solver for compressed-sparse-column matrices, [1, 11].
- The SUPERLUMT package, a threaded linear solver for compressed-sparse-column matrices, [2, 23, 12].

The *spils* family contains the following generic linear solvers:

- The SPGMR package, a solver for the scaled preconditioned GMRES method.
- The SPFGMR package, a solver for the scaled preconditioned Flexible GMRES method.
- The SPBCG package, a solver for the scaled preconditioned Bi-CGStab method.
- The SPTFQMR package, a solver for the scaled preconditioned TFQMR method.

For reasons related to installation, the names of the files involved in these packages begin with the prefix `sundials_`. But despite this, each of the *dls* and *spils* solvers is in fact generic, in that it is usable completely independently of SUNDIALS.

For the sake of space, the functions for the `dense` and `band` modules that work with a matrix type, and the functions in the SPGMR, SPFGMR, SPBCG, and SPTFQMR modules are only summarized briefly, since they are less likely to be of direct use in connection with a SUNDIALS solver. However, the

functions for dense matrices treated as simple arrays and sparse matrices are fully described, because we expect that they will be useful in the implementation of preconditioners used with the combination of one of the SUNDIALS solvers and one of the *spils* linear solvers.

10.1 The DLS modules: DENSE and BAND

The files comprising the DENSE generic linear solver, and their locations in the SUNDIALS *srcdir*, are as follows:

- header files (located in *srcdir*/include/sundials)
  sundials_direct.h, sundials_dense.h,
  sundials_types.h, sundials_math.h, sundials_config.h
- source files (located in *srcdir*/src/sundials)
  sundials_direct.c, sundials_dense.c, sundials_math.c

The files comprising the BAND generic linear solver are as follows:

- header files (located in *srcdir*/include/sundials)
  sundials_direct.h, sundials_band.h,
  sundials_types.h, sundials_math.h, sundials_config.h
- source files (located in *srcdir*/src/sundials)
  sundials_direct.c, sundials_band.c, sundials_math.c

Only two of the preprocessing directives in the header file sundials_config.h are relevant to the DENSE and BAND packages by themselves.

- (required) definition of the precision of the SUNDIALS type realtype. One of the following lines must be present:
  #define SUNDIALS_DOUBLE_PRECISION 1
  #define SUNDIALS_SINGLE_PRECISION 1
  #define SUNDIALS_EXTENDED_PRECISION 1
- (optional) use of generic math functions: #define SUNDIALS_USE_GENERIC_MATH 1

The sundials_types.h header file defines the SUNDIALS realtype and booleantype types and the macro RCONST, while the sundials_math.h header file is needed for the macros SUNMIN and SUNMAX, and the function SUNRabs.

The files listed above for either module can be extracted from the SUNDIALS *srcdir* and compiled by themselves into a separate library or into a larger user code.

10.1.1 Type DlsMat

The type DlsMat, defined in sundials_direct.h is a pointer to a structure defining a generic matrix, and is used with all linear solvers in the *dls* family:

```
typedef struct _DlsMat {
  int type;
  long int M;
  long int N;
  long int ldim;
  long int mu;
  long int ml;
  long int s_mu;
  realtype *data;
  long int ldata;
  realtype **cols;
} *DlsMat;
```

For the DENSE module, the relevant fields of this structure are as follows. Note that a dense matrix of type `DlsMat` need not be square.

type - `SUNDIALS_DENSE` (=1)

M - number of rows

N - number of columns

ldim - leading dimension (`ldim` $\geq$ `M`)

data - pointer to a contiguous block of `realtype` variables

ldata - length of the data array (= `ldim`·`N`). The (`i`,`j`)-th element of a dense matrix `A` of type `DlsMat` (with $0 \leq$ `i` $<$ `M` and $0 \leq$ `j` $<$ `N`) is given by the expression `(A->data)[0][j*M+i]`

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 matrix `A` of type `DlsMat` (with $0 \leq$ `i` $<$ `M` and $0 \leq$ `j` $<$ `N`) is given by the expression `(A->cols)[j][i]`

For the BAND module, the relevant fields of this structure are as follows (see Figure 10.1 for a diagram of the underlying data representation in a banded matrix of type `DlsMat`). Note that only square band matrices are allowed.

type - `SUNDIALS_BAND` (=2)

M - number of rows

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

mu - upper half-bandwidth, $0 \leq$ `mu` $<$ min(`M`,`N`)

ml - lower half-bandwidth, $0 \leq$ `ml` $<$ min(`M`,`N`)

s_mu - storage upper bandwidth, `mu` $\leq$ `s_mu` $<$ `N`. The LU decomposition routine writes the LU factors into the storage for A. The upper triangular factor U, however, may have an upper bandwidth as big as min(`N`-1,`mu`+`ml`) because of partial pivoting. The `s_mu` field holds the upper half-bandwidth allocated for A.

ldim - leading dimension (`ldim` $\geq$ `s_mu`)

data - pointer to a contiguous block of `realtype` variables. The elements of a banded matrix of type `DlsMat` 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`·(`s_mu`+`ml`+1)

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, j−`mu` $\leq i \leq j$+`ml`.

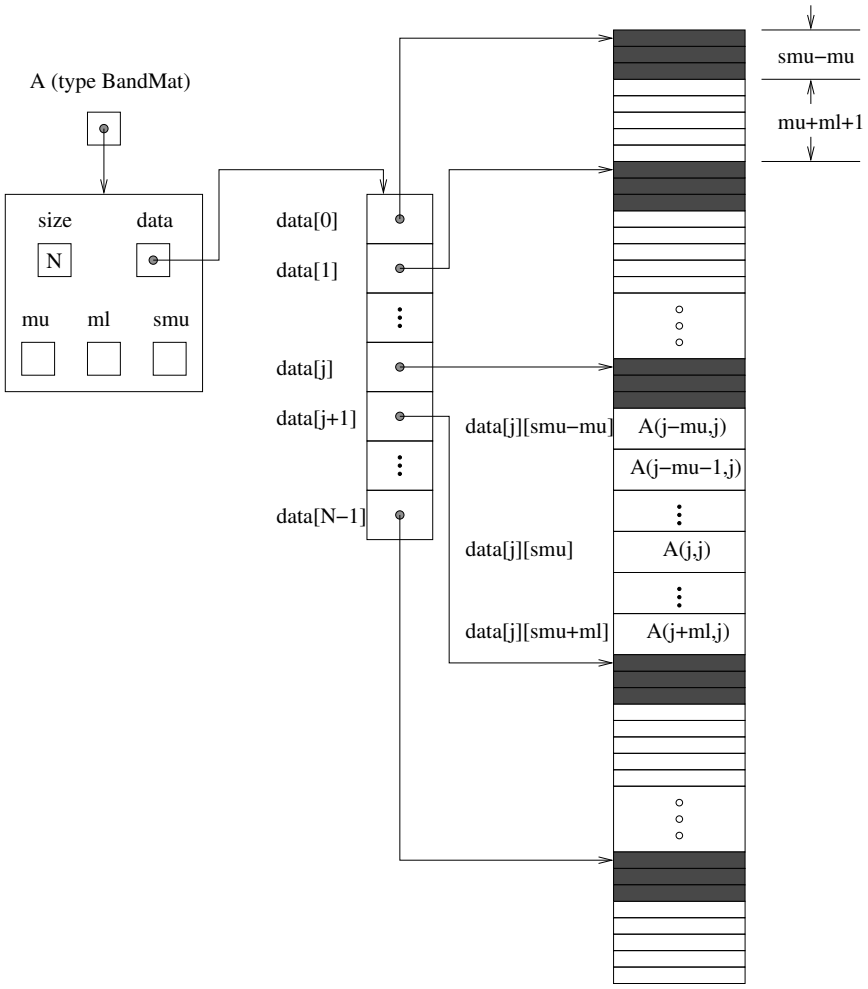


Figure 10.1: Diagram of the storage for a banded matrix of type `DlsMat`. Here `A` is an $N \times N$ band matrix of type `DlsMat` with upper and lower half-bandwidths `mu` and `ml`, respectively. The rows and columns of `A` are numbered from 0 to $N-1$ and the (i,j)-th element of `A` is denoted `A(i,j)`. The greyed out areas of the underlying component storage are used by the `BandGBTRF` and `BandGBTRS` routines.

10.1.2 Accessor macros for the DLS modules

The macros below allow a user to efficiently access individual matrix elements without writing out explicit data structure references and without knowing too much about the underlying element storage. The only storage assumption needed is that elements are stored columnwise and that a pointer to the j-th column of elements can be obtained via the DENSE_COL or BAND_COL macros. Users should use these macros whenever possible.

The following two macros are defined by the DENSE module to provide access to data in the DlsMat type:

- DENSE_ELEM

  Usage : DENSE_ELEM(A,i,j) = a_ij; or a_ij = DENSE_ELEM(A,i,j);

  DENSE_ELEM references the (i,j)-th element of the $M \times N$ DlsMat A, $0 \le$ i $< M$, $0 \le$ j $< N$.

- DENSE_COL

  Usage : col_j = DENSE_COL(A,j);

  DENSE_COL references the j-th column of the $M \times N$ DlsMat A, $0 \le$ j $< N$. The type of the expression DENSE_COL(A,j) is realtype * . After the assignment in the usage above, col_j may be treated as an array indexed from 0 to $M - 1$. The (i, j)-th element of A is referenced by col_j[i].

The following three macros are defined by the BAND module to provide access to data in the DlsMat type:

- BAND_ELEM

  Usage : BAND_ELEM(A,i,j) = a_ij; or a_ij = BAND_ELEM(A,i,j);

  BAND_ELEM references the (i,j)-th element of the $N \times N$ band matrix A, where $0 \le$ i, j $\le N - 1$. The location (i,j) should further satisfy j−(A->mu) $\le$ i $\le$ j+(A->ml).

- BAND_COL

  Usage : col_j = BAND_COL(A,j);

  BAND_COL references the diagonal element of the j-th column of the $N \times N$ band matrix A, $0 \le$ j $\le N - 1$. The type of the expression BAND_COL(A,j) is realtype *. The pointer returned by the call BAND_COL(A,j) can be treated as an array which is indexed from −(A->mu) to (A->ml).

- BAND_COL_ELEM

  Usage : BAND_COL_ELEM(col_j,i,j) = a_ij; or a_ij = BAND_COL_ELEM(col_j,i,j);

  This macro references the (i,j)-th entry of the band matrix A when used in conjunction with BAND_COL to reference the j-th column through col_j. The index (i,j) should satisfy j−(A->mu) $\le$ i $\le$ j+(A->ml).

10.1.3 Functions in the DENSE module

The DENSE module defines two sets of functions with corresponding names. The first set contains functions (with names starting with a capital letter) that act on dense matrices of type DlsMat. The second set contains functions (with names starting with a lower case letter) that act on matrices represented as simple arrays.

The following functions for DlsMat dense matrices are available in the DENSE package. For full details, see the header files sundials_direct.h and sundials_dense.h.

- NewDenseMat: allocation of a DlsMat dense matrix;

- DestroyMat: free memory for a DlsMat matrix;

- `PrintMat`: print a `DlsMat` matrix to standard output.
- `NewIndexArray`: allocation of an array of integers for use as pivots with `DenseGETRF` and `DenseGETRS`;
- `NewIntArray`: allocation of an array of `int` integers for use as pivots with the Lapack dense solvers;
- `NewRealArray`: allocation of an array of `realtype` for use as right-hand side with `DenseGETRS`;
- `DestroyArray`: free memory for an array;
- `SetToZero`: load a matrix with zeros;
- `AddIdentity`: increment a square matrix by the identity matrix;
- `DenseCopy`: copy one matrix to another;
- `DenseScale`: scale a matrix by a scalar;
- `DenseGETRF`: LU factorization with partial pivoting;
- `DenseGETRS`: solution of $Ax = b$ using LU factorization (for square matrices A);
- `DensePOTRF`: Cholesky factorization of a real symmetric positive matrix;
- `DensePOTRS`: solution of $Ax = b$ using the Cholesky factorization of A;
- `DenseGEQRF`: QR factorization of an $m \times n$ matrix, with $m \geq n$;
- `DenseORMQR`: compute the product $w = Qv$, with Q calculated using `DenseGEQRF`;
- `DenseMatvec`: compute the product $y = Ax$, for an M by N matrix A;

The following functions for small dense matrices are available in the DENSE package:

- `newDenseMat`

  `newDenseMat(m,n)` allocates storage for an `m` by `n` dense matrix. It returns a pointer to the newly allocated storage if successful. If the memory request cannot be satisfied, then `newDenseMat` returns `NULL`. The underlying type of the dense matrix returned is `realtype**`. If we allocate a dense matrix `realtype** a` by `a = newDenseMat(m,n)`, then `a[j][i]` references the (`i`,`j`)-th element of the matrix `a`, $0 \leq$ `i` $<$ `m`, $0 \leq$ `j` $< n$, and `a[j]` is a pointer to the first element in the `j`-th column of `a`. The location `a[0]` contains a pointer to `m` $\times$ `n` contiguous locations which contain the elements of `a`.

- `destroyMat`

  `destroyMat(a)` frees the dense matrix `a` allocated by `newDenseMat`;

- `newIndexArray`

  `newIndexArray(n)` allocates an array of `n` integers, all `indextype`. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

- `newIntArray`

  `newIntArray(n)` allocates an array of `n` integers, all `int`. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

- `newRealArray`

  `newRealArray(n)` allocates an array of `n realtype` values. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

- `destroyArray`

  `destroyArray(p)` frees the array `p` allocated by `newIndexArray`, `newIntArray`, or `newRealArray`;

- `denseCopy`

  `denseCopy(a,b,m,n)` copies the `m` by `n` dense matrix `a` into the `m` by `n` dense matrix `b`;

- `denseScale`

  `denseScale(c,a,m,n)` scales every element in the `m` by `n` dense matrix `a` by the scalar `c`;

- `denseAddIdentity`

  `denseAddIdentity(a,n)` increments the *square* `n` by `n` dense matrix `a` by the identity matrix I_n;

- `denseGETRF`

  `denseGETRF(a,m,n,p)` factors the `m` by `n` dense matrix `a`, using Gaussian elimination with row pivoting. It overwrites the elements of `a` with its LU factors and keeps track of the pivot rows chosen in the pivot array `p`.

  A successful LU factorization leaves the matrix `a` and the pivot array `p` with the following information:

  1. `p[k]` contains the row number of the pivot element chosen at the beginning of elimination step `k`, `k` $= 0, 1, ...,$`n`-1.
  2. If the unique LU factorization of `a` is given by $Pa = LU$, where P is a permutation matrix, L is an `m` by `n` lower trapezoidal matrix with all diagonal elements equal to 1, and U is an `n` by `n` upper triangular matrix, then the upper triangular part of `a` (including its diagonal) contains U and the strictly lower trapezoidal part of `a` contains the multipliers, $I - L$. If `a` is square, L is a unit lower triangular matrix.

     `denseGETRF` returns 0 if successful. Otherwise it encountered a zero diagonal element during the factorization, indicating that the matrix `a` does not have full column rank. In this case it returns the column index (numbered from one) at which it encountered the zero.

- `denseGETRS`

  `denseGETRS(a,n,p,b)` solves the `n` by `n` linear system $ax = b$. It assumes that `a` (of size `n` $\times$ `n`) has been LU-factored and the pivot array `p` has been set by a successful call to `denseGETRF(a,n,n,p)`. The solution x is written into the `b` array.

- `densePOTRF`

  `densePOTRF(a,m)` calculates the Cholesky decomposition of the `m` by `m` dense matrix `a`, assumed to be symmetric positive definite. Only the lower triangle of `a` is accessed and overwritten with the Cholesky factor.

- `densePOTRS`

  `densePOTRS(a,m,b)` solves the `m` by `m` linear system $ax = b$. It assumes that the Cholesky factorization of `a` has been calculated in the lower triangular part of `a` by a successful call to `densePOTRF(a,m)`.

- `denseGEQRF`

  `denseGEQRF(a,m,n,beta,wrk)` calculates the QR decomposition of the `m` by `n` matrix `a` (`m` $\geq$ `n`) using Householder reflections. On exit, the elements on and above the diagonal of `a` contain the `n` by `n` upper triangular matrix `R`; the elements below the diagonal, with the array `beta`, represent the orthogonal matrix `Q` as a product of elementary reflectors. The real array `wrk`, of length `m`, must be provided as temporary workspace.

- `denseORMQR`

  `denseORMQR(a,m,n,beta,v,w,wrk)` calculates the product $w = Qv$ for a given vector `v` of length `n`, where the orthogonal matrix Q is encoded in the `m` by `n` matrix `a` and the vector `beta` of length `n`, after a successful call to `denseGEQRF(a,m,n,beta,wrk)`. The real array `wrk`, of length `m`, must be provided as temporary workspace.

- `denseMatvec`

  `denseMatvec(a,x,y,m,n)` calculates the product $y = ax$ for a given vector `x` of length `n`, and `m` by `n` matrix `a`.

10.1.4 Functions in the BAND module

The BAND module defines two sets of functions with corresponding names. The first set contains functions (with names starting with a capital letter) that act on band matrices of type `DlsMat`. The second set contains functions (with names starting with a lower case letter) that act on matrices represented as simple arrays.

The following functions for `DlsMat` banded matrices are available in the BAND package. For full details, see the header files `sundials_direct.h` and `sundials_band.h`.

- `NewBandMat`: allocation of a `DlsMat` band matrix;
- `DestroyMat`: free memory for a `DlsMat` matrix;
- `PrintMat`: print a `DlsMat` matrix to standard output.
- `NewIndexArray`: allocation of an array of integers for use as pivots with `BandGBRF` and `BandGBRS`;
- `NewIntArray`: allocation of an array of `int` integers for use as pivots with the Lapack band solvers;
- `NewRealArray`: allocation of an array of `realtype` for use as right-hand side with `BandGBRS`;
- `DestroyArray`: free memory for an array;
- `SetToZero`: load a matrix with zeros;
- `AddIdentity`: increment a square matrix by the identity matrix;
- `BandCopy`: copy one matrix to another;
- `BandScale`: scale a matrix by a scalar;
- `BandGBTRF`: LU factorization with partial pivoting;
- `BandGBTRS`: solution of $Ax = b$ using LU factorization;
- `BandMatvec`: compute the product $y = Ax$, for a square band matrix A;

The following functions for small band matrices are available in the BAND package:

- `newBandMat`

  `newBandMat(n, smu, ml)` allocates storage for an `n` by `n` band matrix with lower half-bandwidth `ml`.

- `destroyMat`

  `destroyMat(a)` frees the band matrix `a` allocated by `newBandMat`;

- `newIndexArray`

  `newIndexArray(n)` allocates an array of `n` integers, all `long int`. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

- newIntArray

  newIntArray(n) allocates an array of n integers, all int. It returns a pointer to the first element in the array if successful. It returns NULL if the memory request could not be satisfied.

- newRealArray

  newRealArray(n) allocates an array of n realtype values. It returns a pointer to the first element in the array if successful. It returns NULL if the memory request could not be satisfied.

- destroyArray

  destroyArray(p) frees the array p allocated by newIndexArray, newIntArray, or newRealArray;

- bandCopy

  bandCopy(a,b,n,a_smu, b_smu,copymu, copyml) copies the n by n band matrix a into the n by n band matrix b;

- bandScale

  bandScale(c,a,n,mu,ml,smu) scales every element in the n by n band matrix a by c;

- bandAddIdentity

  bandAddIdentity(a,n,smu) increments the n by n band matrix a by the identity matrix;

- bandGETRF

  bandGETRF(a,n,mu,ml,smu,p) factors the n by n band matrix a, using Gaussian elimination with row pivoting. It overwrites the elements of a with its LU factors and keeps track of the pivot rows chosen in the pivot array p.

- bandGETRS

  bandGETRS(a,n,smu,ml,p,b) solves the n by n linear system $ax = b$. It assumes that a (of size n $\times$ n) has been LU-factored and the pivot array p has been set by a successful call to bandGETRF(a,n,mu,ml,smu,p). The solution x is written into the b array.

- bandMatvec

  bandMatvec(a,x,y,n,mu,ml,smu) calculates the product $y = ax$ for a given vector x of length n, and n by n band matrix a.

10.2 The SLS module

SUNDIALS provides a compressed-sparse-column matrix type and sparse matrix support functions. In addition, SUNDIALS provides interfaces to the publically available KLU and SuperLU_MT sparse direct solver packages. The files comprising the SLS matrix module, used in the KLU and SUPERLUMT linear solver packages, and their locations in the SUNDIALS *srcdir*, are as follows:

- header files (located in *srcdir*/include/sundials)
  sundials_sparse.h, sundials_klu_impl.h,
  sundials_superlumt_impl.h, sundials_types.h,
  sundials_math.h, sundials_config.h

- source files (located in *srcdir*/src/sundials)
  sundials_sparse.c, sundials_math.c

Only two of the preprocessing directives in the header file sundials_config.h are relevant to the SLS package by itself:

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  ```
  #define SUNDIALS_DOUBLE_PRECISION 1
  #define SUNDIALS_SINGLE_PRECISION 1
  #define SUNDIALS_EXTENDED_PRECISION 1
  ```
- (optional) use of generic math functions: `#define SUNDIALS_USE_GENERIC_MATH 1`

The `sundials_types.h` header file defines the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the macros `SUNMIN` and `SUNMAX`, and the function `SUNRabs`.

10.2.1 Type SlsMat

SUNDIALS supports operations with matrices either in compressed-sparse-column (CSC) format or in compressed-sparse-row (CSR) format. For convenience, integer sparse matrix identifiers are defined as:

```
#define CSC_MAT 0
#define CSR_MAT 1
```

The type `SlsMat`, defined in `sundials_sparse.h` is a pointer to a structure defining generic CSC and CSR matrix formats, and is used with all linear solvers in the *sls* family:

```
typedef struct _SlsMat {
  int M;
  int N;
  int NNZ;
  int NP;
  realtype *data;
  int sparsetype;
  int *indexvals;
  int *indexptrs;
  int **rowvals;
  int **colptrs;
  int **colvals;
  int **rowptrs;
} *SlsMat;
```

The fields of this structure are as follows (see Figure 10.2 for a diagram of the underlying compressed-sparse-column representation in a sparse matrix of type `SlsMat`). Note that a sparse matrix of type `SlsMat` need not be square.

M - number of rows

N - number of columns

NNZ - maximum number of nonzero entries in the matrix (allocated length of `data` and `rowvals` 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 by the `SparseNewMat` function, 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 a `SlsMat` 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};
rowvals = &indexvals;
colptrs = &indexptrs;
colvals = NULL;
rowptrs = NULL;
```

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. The work associated with operations on the sparse matrix is proportional to this value and so one should use the best understanding of the number of nonzeroes here.

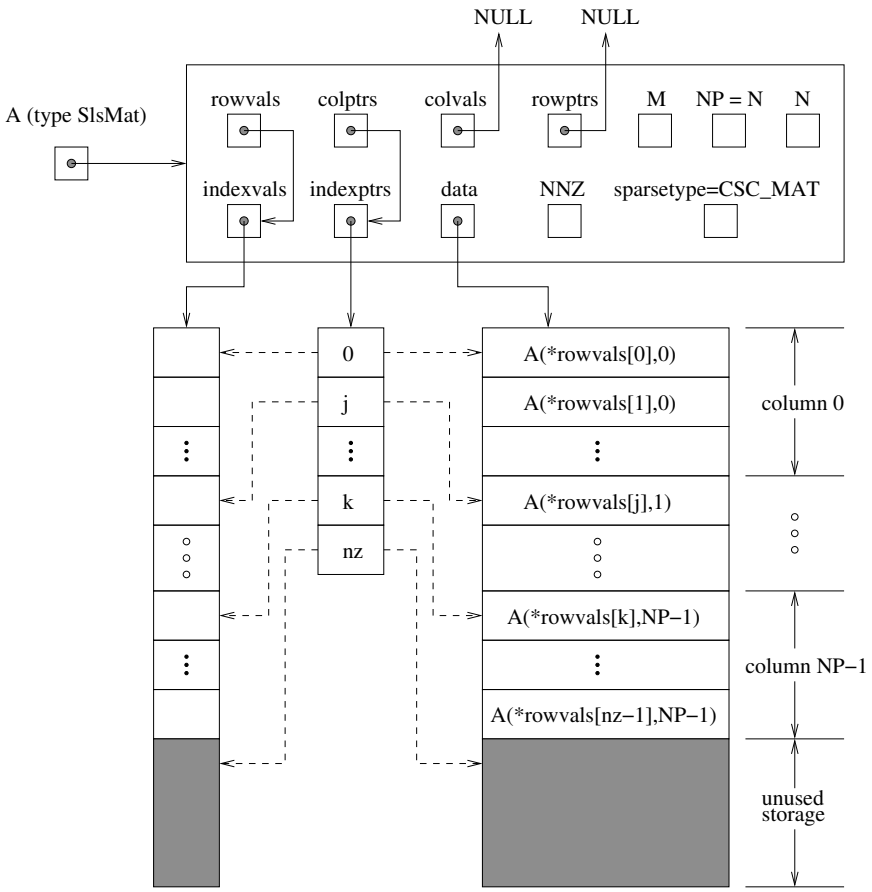


Figure 10.2: Diagram of the storage for a compressed-sparse-column matrix of type `SlsMat`. Here `A` is an $M \times N$ sparse matrix of type `SlsMat` with storage for up to `NNZ` nonzero entries (the allocated length of both `data` and `indexvals`). The entries in `indexvals` may assume values from 0 to $M-1$, corresponding to the row index (zero-based) of each nonzero value. The entries in `data` contain the values of the nonzero entries, with the row i, column j entry of `A` (again, zero-based) denoted as `A(i,j)`. The `indexptrs` array contains $N+1$ entries; the first N denote the starting index of each column within the `indexvals` and `data` arrays, while the final entry points one past the final nonzero entry. Here, although `NNZ` values are allocated, only `nz` are actually filled in; the greyed-out portions of `data` and `indexvals` indicate extra allocated space.

10.2.2 Functions in the SLS module

The SLS module defines functions that act on sparse matrices of type `SlsMat`. For full details, see the header file `sundials_sparse.h`.

- `SparseNewMat`

  `SparseNewMat(M, N, NNZ, sparsetype)` allocates storage for an `M` by `N` sparse matrix, with storage for up to `NNZ` nonzero entries and `sparsetype` storage type (`CSC_MAT` or `CSR_MAT`).

- `SparseFromDenseMat`

  `SparseFromDenseMat(A)` converts a dense or band matrix `A` of type `DlsMat` into a new CSC matrix of type `SlsMat` by retaining only the nonzero values of the matrix `A`.

- `SparseDestroyMat`

  `SparseDestroyMat(A)` frees the memory for a sparse matrix `A` allocated by either `SparseNewMat` or `SparseFromDenseMat`.

- `SparseSetMatToZero(A)` zeros out the `SlsMat` matrix `A`. The storage for `A` is left unchanged.

- `SparseCopyMat`

  `SparseCopyMat(A, B)` copies the `SlsMat A` into the `SlsMat B`. It is assumed that the matrices have the same row/column dimensions and storage type. If `B` has insufficient storage to hold all the nonzero entries of `A`, the data and index arrays in `B` are reallocated to match those in `A`.

- `SparseScaleMat`

  `SparseScaleMat(c, A)` scales every element in the `SlsMat A` by the `realtype` scalar `c`.

- `SparseAddIdentityMat`

  `SparseAddIdentityMat(A)` increments the `SlsMat A` by the identity matrix. If `A` is not square, only the existing diagonal values are incremented. Resizes the `data` and `rowvals` arrays of `A` to allow for new nonzero entries on the diagonal.

- `SparseAddMat`

  `SparseAddMat(A, B)` adds two `SlsMat` matrices `A` and `B`, placing the result back in `A`. Resizes the `data` and `rowvals` arrays of `A` upon completion to exactly match the nonzero storage for the result. Upon successful completion, the return value is zero; otherwise -1 is returned. It is assumed that both matrices have the same size and storage type.

- `SparseReallocMat`

  `SparseReallocMat(A)` eliminates unused storage in the `SlsMat A` by resizing the internal `data` and `rowvals` arrays to contain exactly `colptrs[N]` values.

- `SparseMatvec`

  `SparseMatvec(A, x, y)` computes the sparse matrix-vector product, $y = Ax$. If the `SlsMat A` is a sparse matrix of dimension $M \times N$, then it is assumed that `x` is a `realtype` array of length N, and `y` is a `realtype` array of length M. Upon successful completion, the return value is zero; otherwise -1 is returned.

- `SparsePrintMat`

  `SparsePrintMat(A)` Prints the `SlsMat` matrix `A` to standard output.

10.2.3 The KLU solver

KLU is a sparse matrix factorization and solver library written by Tim Davis [1, 11]. KLU 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). Note that SUNDIALS uses the COLAMD ordering by default with KLU.

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.

The KLU interface in SUNDIALS will perform the symbolic factorization once. It then calls the numerical factorization once and will call the refactor routine until estimates of the numerical conditioning suggest a new factorization should be completed. The KLU interface also has a `ReInit` routine that can be used to force a full refactorization at the next solver setup call.

In order to use the SUNDIALS 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).

Designed for serial calculations only, KLU is supported for calculations employing SUNDIALS' serial or shared-memory parallel NVECTOR modules (see Sections 6.1, 6.3 and 6.4 for details).

10.2.4 The SUPERLUMT solver

SUPERLUMT is a threaded sparse matrix factorization and solver library written by X. Sherry Li [2, 23, 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 SUNDIALS 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).

Designed for serial and threaded calculations only, SUPERLUMT is supported for calculations employing SUNDIALS' serial or shared-memory parallel NVECTOR modules (see Sections 6.1, 6.3 and 6.4 for details).

10.3 The SPILS modules: SPGMR, SPFGMR, SPBCG, and SPTFQMR

The *spils* modules contain implementations of some of the most commonly use scaled preconditioned Krylov solvers. A linear solver module from the *spils* family can be used in conjunction with any NVECTOR implementation library.

10.3.1 The SPGMR module

The SPGMR package, in the files `sundials_spgmr.h` and `sundials_spgmr.c`, includes an implementation of the scaled preconditioned GMRES method. A separate code module, implemented in `sundials_iterative.(h,c)`, contains auxiliary functions that support SPGMR, as well as the other Krylov solvers in SUNDIALS (SPFGMR, SPBCG, and SPTFQMR). For full details, including usage instructions, see the header files `sundials_spgmr.h` and `sundials_iterative.h`.

The files comprising the SPGMR generic linear solver, and their locations in the SUNDIALS *srcdir*, are as follows:

- header files (located in *srcdir*`/include/sundials`)
  `sundials_spgmr.h`, `sundials_iterative.h`, `sundials_nvector.h`,

sundials_types.h, sundials_math.h, sundials_config.h

- source files (located in *srcdir*/src/sundials)
  sundials_spgmr.c, sundials_iterative.c, sundials_nvector.c

Only two of the preprocessing directives in the header file sundials_config.h are required to use the SPGMR package by itself:

- (required) definition of the precision of the SUNDIALS type realtype. One of the following lines must be present:
  ```
  #define SUNDIALS_DOUBLE_PRECISION 1
  #define SUNDIALS_SINGLE_PRECISION 1
  #define SUNDIALS_EXTENDED_PRECISION 1
  ```
- (optional) use of generic math functions:
  ```
  #define SUNDIALS_USE_GENERIC_MATH 1
  ```

The sundials_types.h header file defines the SUNDIALS realtype and booleantype types and the macro RCONST, while the sundials_math.h header file is needed for the macros SUNMIN, SUNMAX, and SUNSQR, and the functions SUNRabs and SUNRsqrt.

The generic NVECTOR files, sundials_nvector.(h,c) are needed for the definition of the generic N_Vector type and functions. The NVECTOR functions used by the SPGMR module are: N_VDotProd, N_VLinearSum, N_VScale, N_VProd, N_VDiv, N_VConst, N_VClone, N_VCloneVectorArray, N_VDestroy, and N_VDestroyVectorArray.

The nine files listed above can be extracted from the SUNDIALS *srcdir* and compiled by themselves into an SPGMR library or into a larger user code.

The following functions are available in the SPGMR package:

- SpgmrMalloc: allocation of memory for SpgmrSolve;
- SpgmrSolve: solution of $Ax = b$ by the SPGMR method;
- SpgmrFree: free memory allocated by SpgmrMalloc.

The following functions are available in the support package sundials_iterative.(h,c):

- ModifiedGS: performs modified Gram-Schmidt procedure;
- ClassicalGS: performs classical Gram-Schmidt procedure;
- QRfact: performs QR factorization of Hessenberg matrix;
- QRsol: solves a least squares problem with a Hessenberg matrix factored by QRfact.

10.3.2 The SPFGMR module

The SPFGMR package, in the files sundials_spfgmr.h and sundials_spfgmr.c, includes an implementation of the scaled preconditioned Flexible GMRES method. For full details, including usage instructions, see the file sundials_spfgmr.h.

The files needed to use the SPFGMR module by itself are the same as for the SPGMR module, but with sundials_spfgmr.(h,c) in place of sundials_spgmr.(h,c).

The following functions are available in the SPFGMR package:

- SpfgmrMalloc: allocation of memory for SpfgmrSolve;
- SpfgmrSolve: solution of $Ax = b$ by the SPFGMR method;
- SpfgmrFree: free memory allocated by SpfgmrMalloc.

10.3.3 The SPBCG module

The SPBCG package, in the files `sundials_spbcgs.h` and `sundials_spbcgs.c`, includes an implementation of the scaled preconditioned Bi-CGStab method. For full details, including usage instructions, see the file `sundials_spbcgs.h`.

The files needed to use the SPBCG module by itself are the same as for the SPGMR module, but with `sundials_spbcgs.(h,c)` in place of `sundials_spgmr.(h,c)`.

The following functions are available in the SPBCG package:

- `SpbcgMalloc`: allocation of memory for `SpbcgSolve`;
- `SpbcgSolve`: solution of $Ax = b$ by the SPBCG method;
- `SpbcgFree`: free memory allocated by `SpbcgMalloc`.

10.3.4 The SPTFQMR module

The SPTFQMR package, in the files `sundials_sptfqmr.h` and `sundials_sptfqmr.c`, includes an implementation of the scaled preconditioned TFQMR method. For full details, including usage instructions, see the file `sundials_sptfqmr.h`.

The files needed to use the SPTFQMR module by itself are the same as for the SPGMR module, but with `sundials_sptfqmr.(h,c)` in place of `sundials_spgmr.(h,c)`.

The following functions are available in the SPTFQMR package:

- `SptfqmrMalloc`: allocation of memory for `SptfqmrSolve`;
- `SptfqmrSolve`: solution of $Ax = b$ by the SPTFQMR method;
- `SptfqmrFree`: free memory allocated by `SptfqmrMalloc`.

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 on the installation procedure begins with a few common observations:

- The remainder of this chapter will follow these conventions:

  srcdir 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 *srcdir* 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 *srcdir*.

- 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 `2.8.1` or higher and a working 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. While many Linux distributions offer CMake, the version included is probably out of date. Many new CMake features have been added recently, and you should download the latest version from `http://www.cmake.org`. Build instructions for CMake (only necessary for Unix-like systems) can be found on the CMake website. Once CMake is installed, Linux/Unix users will be able to use `ccmake`, 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 *srcdir*:

```
% ccmake ../srcdir
```

The default configuration screen is shown in Figure A.1.

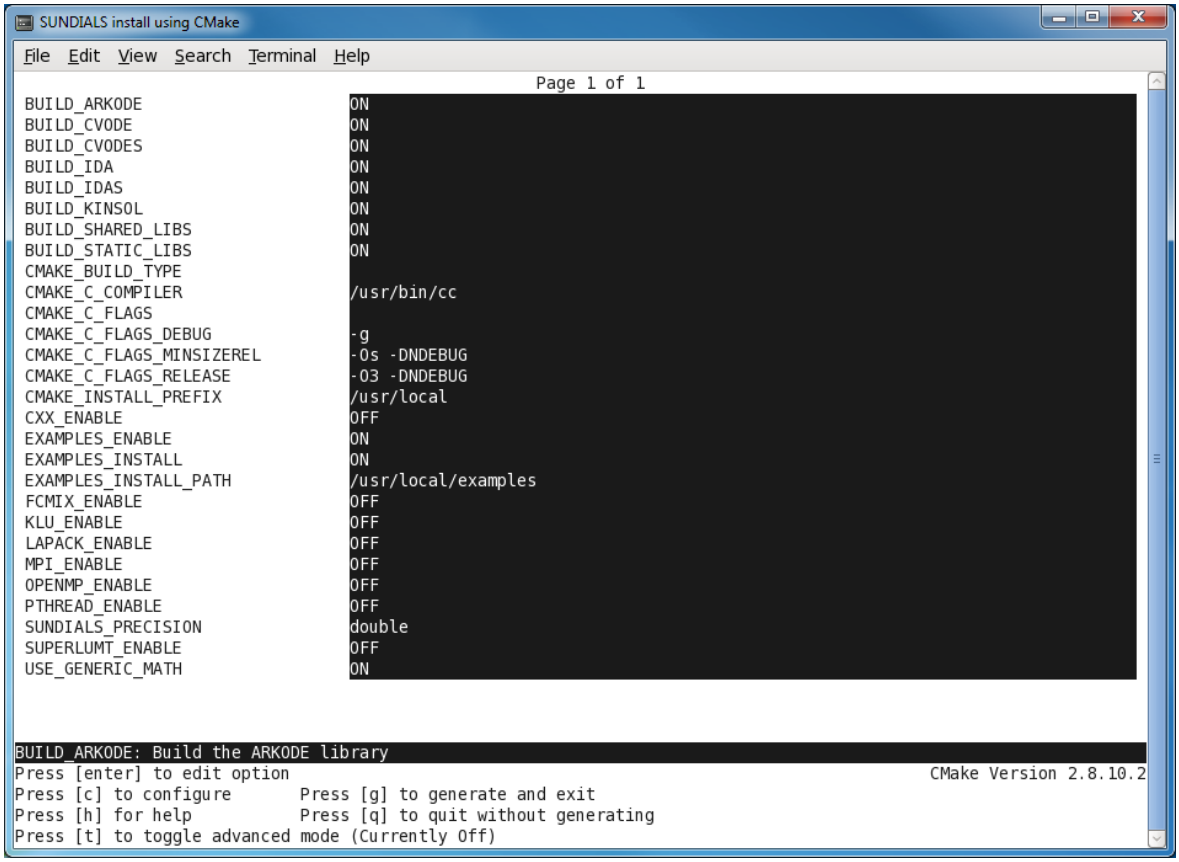


Figure A.1: Default configuration screen. Note: Initial screen is empty. To get this default configuration, press 'c' repeatedly (accepting default values denoted with asterisk) until the 'g' option is available.

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
```

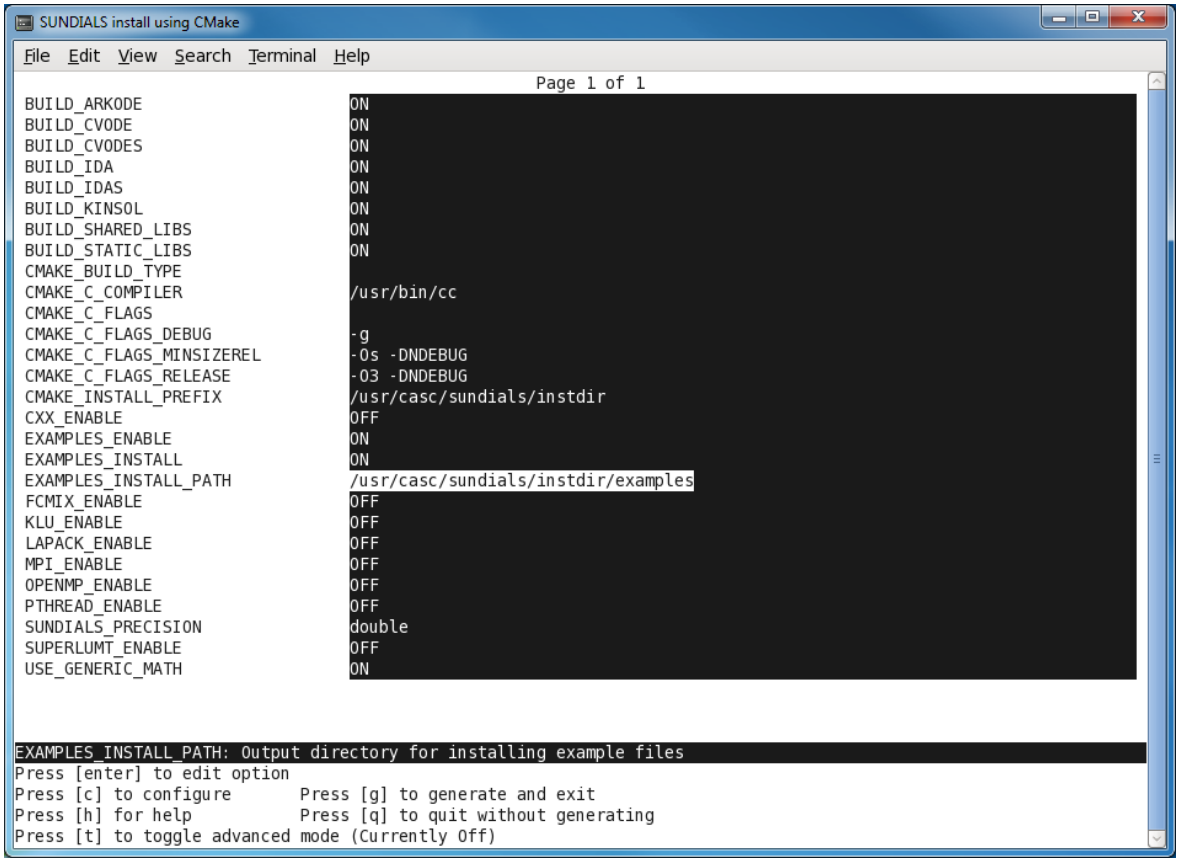


Figure A.2: Changing the *instdir* for SUNDIALS and corresponding examples

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 \
> ../srcdir
% make
% make install
```

A.1.2 Configuration options (Unix/Linux)

A complete list of all available options for a CMake-based SUNDIALS configuration is provide below. Note that the default values shown are for a typical configuration on a Linux system and are provided as illustration only.

`BUILD_ARKODE` - Build the ARKODE library
Default: ON

`BUILD_CVODE` - Build the CVODE library
Default: ON

`BUILD_CVODES` - Build the CVODES library
Default: ON

BUILD_IDA - Build the IDA library
Default: ON

BUILD_IDAS - Build the IDAS library
Default: ON

BUILD_KINSOL - Build the KINSOL library
Default: ON

BUILD_SHARED_LIBS - Build shared libraries
Default: OFF

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 MinSizeRel
Default:

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 compiler during debug builds
Default: -g

CMAKE_C_FLAGS_MINSIZEREL - Flags used by the compiler during release minsize builds
Default: -Os -DNDEBUG

CMAKE_C_FLAGS_RELEASE - Flags used by the 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 (LAPACK_ENABLE is ON).

CMAKE_Fortran_FLAGS - Flags for Fortran compiler
Default:

CMAKE_Fortran_FLAGS_DEBUG - Flags used by the compiler during debug builds
Default:

CMAKE_Fortran_FLAGS_MINSIZEREL - Flags used by the compiler during release minsize builds
Default:

CMAKE_Fortran_FLAGS_RELEASE - Flags used by the compiler during release builds
Default:

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.

EXAMPLES_ENABLE - Build the SUNDIALS examples
Default: ON

EXAMPLES_INSTALL - Install example files
Default: ON
Note: This option is triggered only if building example programs is enabled (EXAMPLES_ENABLE 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 an examples subdirectory created under CMAKE_INSTALL_PREFIX.

FCMIX_ENABLE - Enable Fortran-C support
Default: OFF

HYPRE_ENABLE - Enable hypre support
Default: OFF

HYPRE_INCLUDE_DIR - Path to hypre header files

HYPRE_LIBRARY - Path to hypre installed library

KLU_ENABLE - Enable KLU support
Default: OFF

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 the two additional options see below.

LAPACK_LIBRARIES - Lapack (and Blas) libraries
Default: /usr/lib/liblapack.so;/usr/lib/libblas.so
Note: CMake will search for these libraries in your LD_LIBRARY_PATH prior to searching default system paths.

MPI_ENABLE - Enable MPI support
Default: OFF
Note: Setting this option to ON will trigger several additional options related to MPI.

MPI_MPICC - mpicc program
Default:

MPI_RUN_COMMAND - Specify run command for MPI
Default: mpirun
Note: This can either be set to mpirun for OpenMPI or srun if jobs are managed by SLURM - Simple Linux Utility for Resource Management as exists on LLNL's high performance computing clusters.

MPI_MPIF77 - mpif77 program
Default:
Note: This option is triggered only if using MPI compiler scripts (MPI_USE_MPISCRIPTS is ON) and Fortran-C support is enabled (FCMIx_ENABLE is ON).

OPENMP_ENABLE - Enable OpenMP support
Default: OFF
Turn on support for the OpenMP based nvector.

PETSC_ENABLE - Enable PETSc support
Default: OFF

PETSC_INCLUDE_DIR - Path to PETSc header files

PETSC_LIBRARY_DIR - Path to PETSc installed library files

PTHREAD_ENABLE - Enable Pthreads support
Default: OFF
Turn on support for the Pthreads based nvector.

SUNDIALS_PRECISION - Precision used in SUNDIALS, options are: double, single or extended
Default: double

SUPERLUMT_ENABLE - Enable SUPERLU_MT support
Default: OFF

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

USE_GENERIC_MATH - Use generic (stdc) math libraries
Default: ON

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/srcdir
%
% make install
%
```

To disable installation of the examples, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DMPI_ENABLE=ON \
> -DFCMIX_ENABLE=ON \
> -DEXAMPLES_INSTALL=OFF \
> /home/myname/sundials/srcdir
%
% 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.

Building with LAPACK and BLAS

To enable LAPACK and BLAS libraries, set the LAPACK_ENABLE option to ON. If the directory containing the LAPACK and BLAS libraries is in the LD_LIBRARY_PATH environment variable, CMake will set the LAPACK_LIBRARIES variable accordingly, otherwise CMake will attemp to find the LAPACK and BLAS libraries in standard system locations. To explicitly tell CMake what libraries to use, the LAPACK_LIBRARIES varible can be set to the desired libraries. Example:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DLAPACK_LIBRARIES=/mypath/lib/liblapack.so;/mypath/lib/libblas.so \
> /home/myname/sundials/srcdir
%
% make install
%
```

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-scalable-linear-solvers-multigrid-methods. 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 to the lib path of the hypre installation.

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 both EXAMPLES_ENABLE and 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 *srcdir*
2. Create a separate *builddir*
3. Open a Visual Studio Command Prompt and cd to *builddir*
4. Run cmake-gui ../*srcdir*
   (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 Tables A.1 and A.2. 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	
	Header files	sundials/sundials_config.h	sundials/sundials_types.h
		sundials/sundials_math.h	
		sundials/sundials_nvector.h	sundials/sundials_fnvector.h
		sundials/sundials_direct.h	sundials/sundials_lapack.h
		sundials/sundials_dense.h	sundials/sundials_band.h
		sundials/sundials_sparse.h	
		sundials/sundials_iterative.h	sundials/sundials_spgmr.h
		sundials/sundials_spbcgs.h	sundials/sundials_sptfqmr.h
		sundials/sundials_pcg.h	sundials/sundials_spfgmr.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	
CVODE	Libraries	libsundials_cvode.*lib*	libsundials_fcvode.a
	Header files	cvode/cvode.h	cvode/cvode_impl.h
		cvode/cvode_direct.h	cvode/cvode_lapack.h
		cvode/cvode_dense.h	cvode/cvode_band.h
		cvode/cvode_diag.h	
		cvode/cvode_sparse.h	cvode/cvode_klu.h
		cvode/cvode_superlumt.h	
		cvode/cvode_spils.h	cvode/cvode_spgmr.h
		cvode/cvode_sptfqmr.h	cvode/cvode_spbcgs.h
		cvode/cvode_bandpre.h	cvode/cvode_bbdpre.h
CVODES	Libraries	libsundials_cvodes.*lib*	
	Header files	cvodes/cvodes.h	cvodes/cvodes_impl.h
		cvodes/cvodes_direct.h	cvodes/cvodes_lapack.h
		cvodes/cvodes_dense.h	cvodes/cvodes_band.h
		cvodes/cvodes_diag.h	
		cvodes/cvodes_sparse.h	cvodes/cvodes_klu.h
		cvodes/cvodes_superlumt.h	
		cvodes/cvodes_spils.h	cvodes/cvodes_spgmr.h
		cvodes/cvodes_sptfqmr.h	cvodes/cvodes_spbcgs.h
		cvodes/cvodes_bandpre.h	cvodes/cvodes_bbdpre.h
ARKODE	Libraries	libsundials_arkode.*lib*	libsundials_farkode.a
	Header files	arkode/arkode.h	arkode/arkode_impl.h
		arkode/arkode_direct.h	arkode/arkode_lapack.h
		arkode/arkode_dense.h	arkode/arkode_band.h
		arkode/arkode_sparse.h	arkode/arkode_klu.h
		arkode/arkode_superlumt.h	
		arkode/arkode_spils.h	arkode/arkode_spgmr.h
		arkode/arkode_sptfqmr.h	arkode/arkode_spbcgs.h
		arkode/arkode_pcg.h	arkode/arkode_spfgmr.h
		arkode/arkode_bandpre.h	arkode/arkode_bbdpre.h

Table A.2: SUNDIALS libraries and header files (cont.)

IDA	Libraries	libsundials_ida.*lib*	libsundials_fida.a
	Header files	ida/ida.h	ida/ida_impl.h
		ida/ida_direct.h	ida/ida_lapack.h
		ida/ida_dense.h	ida/ida_band.h
		ida/ida_sparse.h	ida/ida_klu.h
		ida/ida_superlumt.h	
		ida/ida_spils.h	ida/ida_spgmr.h
		ida/ida_spbcgs.h	ida/ida_sptfqmr.h
		ida/ida_bbdpre.h	
IDAS	Libraries	libsundials_idas.*lib*	
	Header files	idas/idas.h	idas/idas_impl.h
		idas/idas_direct.h	idas/idas_lapack.h
		idas/idas_dense.h	idas/idas_band.h
		idas/idas_sparse.h	idas/idas_klu.h
		idas/idas_superlumt.h	
		idas/idas_spils.h	idas/idas_spgmr.h
		idas/idas_spbcgs.h	idas/idas_sptfqmr.h
		idas/idas_bbdpre.h	
KINSOL	Libraries	libsundials_kinsol.*lib*	libsundials_fkinsol.a
	Header files	kinsol/kinsol.h	kinsol/kinsol_impl.h
		kinsol/kinsol_direct.h	kinsol/kinsol_lapack.h
		kinsol/kinsol_dense.h	kinsol/kinsol_band.h
		kinsol/kinsol_sparse.h	kinsol/kinsol_klu.h
		kinsol/kinsol_superlumt.h	
		kinsol/kinsol_spils.h	kinsol/kinsol_spgmr.h
		kinsol/kinsol_spbcgs.h	kinsol/kinsol_sptfqmr.h
		kinsol/kinsol_bbdpre.h	kinsol/kinsol_spfgmr.h

Appendix B

CVODE Constants

Below we list all input and output constants used by the main solver and linear solver modules, together with their numerical values and a short description of their meaning.

B.1 CVODE input constants

CVODE **main solver module**		
CV_ADAMS	1	Adams-Moulton linear multistep method.
CV_BDF	2	BDF linear multistep method.
CV_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.
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 CVODE output constants

CVODE **main solver module**		
CV_SUCCESS	0	Successful function return.
CV_TSTOP_RETURN	1	`CVode` succeeded by reaching the specified stopping point.
CV_ROOT_RETURN	2	`CVode` succeeded and found one or more roots.
CV_WARNING	99	`CVode` succeeded but an unusual situation occurred.
CV_TOO_MUCH_WORK	-1	The solver took `mxstep` internal steps but could not reach tout.

CV_TOO_MUCH_ACC	-2	The solver could not satisfy the accuracy demanded by the user for some internal step.
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_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.

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.

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.

CVSLS linear solver module

CVSLS_SUCCESS	0	Successful function return.
CVSLS_MEM_NULL	-1	The cv_mem argument was NULL.
CVSLS_LMEM_NULL	-2	The CVSLS linear solver has not been initialized.
CVSLS_ILL_INPUT	-3	The CVSLS solver is not compatible with the current NVECTOR module or other input is invalid.
CVSLS_MEM_FAIL	-4	A memory allocation request failed.
CVSLS_JAC_NOSET	-5	The Jacobian evaluation routine was not been set before the linear solver setup routine was called.
CVSLS_PACKAGE_FAIL	-6	An external package call return a failure error code.
CVSLS_JACFUNC_UNRECVR	-7	The Jacobian function failed in an unrecoverable manner.
CVSLS_JACFUNC_RECVR	-8	The Jacobian function had a recoverable error.

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.

SPGMR generic linear solver module

SPGMR_SUCCESS	0	Converged.
SPGMR_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPGMR_CONV_FAIL	2	Failure to converge.
SPGMR_QRFACT_FAIL	3	A singular matrix was found during the QR factorization.
SPGMR_PSOLVE_FAIL_REC	4	The preconditioner solve function failed recoverably.
SPGMR_ATIMES_FAIL_REC	5	The Jacobian-times-vector function failed recoverably.
SPGMR_PSET_FAIL_REC	6	The preconditioner setup routine failed recoverably.
SPGMR_MEM_NULL	-1	The SPGMR memory is NULL
SPGMR_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed unrecoverably.
SPGMR_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.

SPGMR_GS_FAIL	-4	Failure in the Gram-Schmidt procedure.
SPGMR_QRSOL_FAIL	-5	The matrix R was found to be singular during the QR solve phase.
SPGMR_PSET_FAIL_UNREC	-6	The preconditioner setup routine failed unrecoverably.

SPFGMR **generic linear solver module (only available in** KINSOL **and** ARKODE**)**

SPFGMR_SUCCESS	0	Converged.
SPFGMR_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPFGMR_CONV_FAIL	2	Failure to converge.
SPFGMR_QRFACT_FAIL	3	A singular matrix was found during the QR factorization.
SPFGMR_PSOLVE_FAIL_REC	4	The preconditioner solve function failed recoverably.
SPFGMR_ATIMES_FAIL_REC	5	The Jacobian-times-vector function failed recoverably.
SPFGMR_PSET_FAIL_REC	6	The preconditioner setup routine failed recoverably.
SPFGMR_MEM_NULL	-1	The SPFGMR memory is NULL
SPFGMR_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed unrecoverably.
SPFGMR_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPFGMR_GS_FAIL	-4	Failure in the Gram-Schmidt procedure.
SPFGMR_QRSOL_FAIL	-5	The matrix R was found to be singular during the QR solve phase.
SPFGMR_PSET_FAIL_UNREC	-6	The preconditioner setup routine failed unrecoverably.

SPBCG **generic linear solver module**

SPBCG_SUCCESS	0	Converged.
SPBCG_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPBCG_CONV_FAIL	2	Failure to converge.
SPBCG_PSOLVE_FAIL_REC	3	The preconditioner solve function failed recoverably.
SPBCG_ATIMES_FAIL_REC	4	The Jacobian-times-vector function failed recoverably.
SPBCG_PSET_FAIL_REC	5	The preconditioner setup routine failed recoverably.
SPBCG_MEM_NULL	-1	The SPBCG memory is NULL
SPBCG_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed unrecoverably.
SPBCG_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPBCG_PSET_FAIL_UNREC	-4	The preconditioner setup routine failed unrecoverably.

SPTFQMR **generic linear solver module**

SPTFQMR_SUCCESS	0	Converged.
SPTFQMR_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPTFQMR_CONV_FAIL	2	Failure to converge.
SPTFQMR_PSOLVE_FAIL_REC	3	The preconditioner solve function failed recoverably.
SPTFQMR_ATIMES_FAIL_REC	4	The Jacobian-times-vector function failed recoverably.
SPTFQMR_PSET_FAIL_REC	5	The preconditioner setup routine failed recoverably.
SPTFQMR_MEM_NULL	-1	The SPTFQMR memory is NULL

SPTFQMR_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed.
SPTFQMR_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPTFQMR_PSET_FAIL_UNREC	-4	The preconditioner setup routine failed unrecoverably.

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. User Documentation for PVODE, An ODE Solver for Parallel Computers. Technical Report UCRL-ID-130884, LLNL, May 1998.

[8] 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.

[9] S. D. Cohen and A. C. Hindmarsh. CVODE User Guide. Technical Report UCRL-MA-118618, LLNL, September 1994.

[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] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14:470–482, 1993.

[14] K. L. Hiebert and L. F. Shampine. Implicitly Defined Output Points for Solutions of ODEs. Technical Report SAND80-0180, Sandia National Laboratories, February 1980.

[15] 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.

[16] A. C. Hindmarsh. Avoiding BDF Stability Barriers in the MOL Solution of Advection-Dominated Problems. *Appl. Num. Math.*, 17:311–318, 1995.

[17] A. C. Hindmarsh. The PVODE and IDA Algorithms. Technical Report UCRL-ID-141558, LLNL, December 2000.

[18] 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.

[19] A. C. Hindmarsh and R. Serban. Example Programs for CVODE v2.9.0. Technical report, LLNL, 2016. UCRL-SM-208110.

[20] A. C. Hindmarsh, R. Serban, and A. Collier. Example Programs for IDA v2.9.0. Technical Report UCRL-SM-208113, LLNL, 2016.

[21] 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.

[22] 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.

[23] X. S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.

[24] 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.

[25] Daniel R. Reynolds. Example Programs for ARKODE v1.1.0. Technical report, Southern Methodist University, 2016.

[26] 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.

[27] 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, 9

BAND generic linear solver
  functions, 170
    small matrix, 170–171
  macros, 167
  type DlsMat, 164–167
BAND_COL, 67, 167
BAND_COL_ELEM, 67, 167
BAND_ELEM, 67, 167
bandAddIdentity, 171
bandCopy, 171
bandGETRF, 171
bandGETRS, 171
bandMatvec, 171
bandScale, 171
BDF method, 9
Bi-CGStab method, 33, 46, 178
BIG_REAL, 22, 102

CLASSICAL_GS, 46
CSC_MAT, 31
CV_ADAMS, 26, 63
CV_BAD_DKY, 48
CV_BAD_K, 48
CV_BAD_T, 48
CV_BDF, 26, 63
CV_CONV_FAILURE, 35
CV_ERR_FAILURE, 35
CV_FIRST_RHSFUNC_ERR, 64
CV_FIRST_RHSFUNC_FAIL, 35
CV_FUNCTIONAL, 26, 41
CV_ILL_INPUT, 26, 27, 34, 38–41, 47, 63
CV_LINIT_FAIL, 35
CV_LSETUP_FAIL, 35, 66–68, 75
CV_LSOLVE_FAIL, 35
CV_MEM_FAIL, 26
CV_MEM_NULL, 26, 27, 34, 37–41, 47, 48, 50–56, 63
CV_NEWTON, 26, 41
CV_NO_MALLOC, 27, 34, 63
CV_NORMAL, 34
CV_ONE_STEP, 34
CV_REPTD_RHSFUNC_ERR, 35
CV_RHSFUNC_FAIL, 35, 64
CV_ROOT_RETURN, 34
CV_RTFUNC_FAIL, 35, 65
CV_SUCCESS, 26, 27, 34, 37–41, 47, 48, 50–56, 63
CV_TOO_CLOSE, 34
CV_TOO_MUCH_ACC, 35
CV_TOO_MUCH_WORK, 34
CV_TSTOP_RETURN, 34
CV_UNREC_RHSFUNC_ERR, 35, 64
CV_WARNING, 64
CVBAND linear solver
  Jacobian approximation used by, 42
  memory requirements, 56
  NVECTOR compatibility, 30
  optional input, 41–42
  optional output, 56–57
  selection of, 30
  use in FCVODE, 84
CVBand, 24, 29, 30, 67
CVBANDPRE preconditioner
  description, 71
  optional output, 72–73
  usage, 71–72
  user-callable functions, 72
CVBandPrecGetNumRhsEvals, 73
CVBandPrecGetWorkSpace, 73
CVBandPrecInit, 72
CVBBDPRE preconditioner
  description, 73–74
  optional output, 77–78
  usage, 75–76
  user-callable functions, 76–77
  user-supplied functions, 74–75
CVBBDPrecGetNumGfnEvals, 78
CVBBDPrecGetWorkSpace, 77
CVBBDPrecInit, 76
CVBBDPrecReInit, 77
CVDENSE linear solver
  Jacobian approximation used by, 41
  memory requirements, 56
  NVECTOR compatibility, 29
  optional input, 41–42
  optional output, 56–57
  selection of, 29

use in FCVODE, 83
CVDense, 24, 29, 65
CVDIAG linear solver
Jacobian approximation used by, 32
memory requirements, 57
optional output, 57–59
selection of, 32
use in FCVODE, 83
CVDiag, 24, 29, 32
CVDIAG_ILL_INPUT, 32
CVDIAG_LMEM_NULL, 58
CVDIAG_MEM_FAIL, 32
CVDIAG_MEM_NULL, 32, 58
CVDIAG_SUCCESS, 32, 58
CVDiagGetLastFlag, 58
CVDiagGetNumRhsEvals, 58
CVDiagGetReturnFlagName, 59
CVDiagGetWorkSpace, 58
CVDLS_ILL_INPUT, 29, 30
CVDLS_JACFUNC_RECVR, 66, 67
CVDLS_JACFUNC_UNRECVR, 66, 67
CVDLS_LMEM_NULL, 42, 56, 57
CVDLS_MEM_FAIL, 29, 30
CVDLS_MEM_NULL, 29, 30, 42, 56, 57
CVDLS_SUCCESS, 29, 30, 42, 56, 57
CVDlsBandJacFn, 67
CVDlsDenseJacFn, 65
CVDlsGetLastFlag, 57
CVDlsGetNumJacEvals, 56
CVDlsGetNumRhsEvals, 57
CVDlsGetReturnFlagName, 57
CVDlsGetWorkSpace, 56
CVDlsSetBandJacFn, 42
CVDlsSetDenseJacFn, 41
CVErrHandlerFn, 64
CVEwtFn, 65
CVKLU, 24, 29, 31, 68
CVKLU linear solver
Jacobian approximation used by, 42
matrix reordering algorithm specification, 43
NVECTOR compatibility, 30
optional input, 42–44
optional output, 59
reinitialization, 43
selection of, 30
CVKLUReInit, 43
CVKLUSetOrdering, 43
CVLapackBand, 24, 29, 30, 67
CVLapackDense, 24, 29, 30, 65
CVODE, 1
motivation for writing in C, 2
package structure, 17
relationship to CVODE, PVODE, 1–2
relationship to VODE, VODPK, 1
CVODE linear solvers
built on generic solvers, 29
CVBAND, 30
CVDENSE, 29
CVDIAG, 32
CVKLU, 30
CVSPBCG, 32
CVSPGMR, 32
CVSPTFQMR, 33
CVSUPERLUMT, 31
header files, 22
implementation details, 20
list of, 17–19
NVECTOR compatibility, 21
selecting one, 29
CVode, 24, 34
cvode.h, 22
cvode_band.h, 22
cvode_dense.h, 22
cvode_diag.h, 22
cvode_klu.h, 22
cvode_lapack.h, 22
cvode_spbcgs.h, 23
cvode_spgmr.h, 23
cvode_sptfqmr.h, 23
cvode_superlumt.h, 23
CVodeCreate, 26
CVodeFree, 25, 26
CVodeGetActualInitStep, 52
CVodeGetCurrentOrder, 51
CVodeGetCurrentStep, 52
CVodeGetCurrentTime, 52
CVodeGetDky, 47
CVodeGetErrWeights, 53
CVodeGetEstLocalErrors, 53
CVodeGetIntegratorStats, 54
CVodeGetLastOrder, 51
CVodeGetLastStep, 52
CVodeGetNonlinSolvStats, 55
CVodeGetNumErrTestFails, 51
CVodeGetNumGEvals, 56
CVodeGetNumLinSolvSetups, 51
CVodeGetNumNonlinSolvConvFails, 54
CVodeGetNumNonlinSolvIters, 54
CVodeGetNumRhsEvals, 50
CVodeGetNumStabLimOrderReds, 53
CVodeGetNumSteps, 50
CVodeGetReturnFlagName, 55
CVodeGetRootInfo, 55
CVodeGetTolScaleFactor, 53
CVodeGetWorkSpace, 48
CVodeInit, 26, 62
CVodeReInit, 62
CVodeRootInit, 33

CVodeSetErrFile, 37
CVodeSetErrHandlerFn, 37
CVodeSetInitStep, 39
CVodeSetIterType, 41
CVodeSetMaxConvFails, 40
CVodeSetMaxErrTestFails, 40
CVodeSetMaxHnilWarns, 38
CVodeSetMaxNonlinIters, 40
CVodeSetMaxNumSteps, 38
CVodeSetMaxOrder, 38
CVodeSetMaxStep, 39
CVodeSetMinStep, 39
CVodeSetNoInactiveRootWarn, 47
CVodeSetNonlinConvCoef, 41
CVodeSetRootDirection, 47
CVodeSetStabLimDet, 38
CVodeSetStopTime, 40
CVodeSetUserData, 37
CVodeSStolerances, 27
CVodeSVtolerances, 27
CVodeWFtolerances, 27
CVRhsFn, 26, 63
CVRootFn, 65
CVSLS_ILL_INPUT, 31, 43, 44
CVSLS_JACFUNC_RECVR, 68
CVSLS_JACFUNC_UNRECVR, 68
CVSLS_LMEM_NULL, 43, 59
CVSLS_MEM_FAIL, 31, 43
CVSLS_MEM_NULL, 31, 43, 44, 59
CVSLS_PACKAGE_FAIL, 31
CVSLS_SUCCESS, 31, 43, 44, 59
CVSlsGetLastFlag, 59
CVSlsGetNumJacEvals, 59
CVSlsGetReturnFlagName, 59
CVSlsSetSparseJacFn, 42
CVSlsSparseJacFn, 68
CVSPARSE linear solver
  use in FCVODE, 85
CVSPBCG linear solver
  Jacobian approximation used by, 44
  memory requirements, 60
  optional input, 44–46
  optional output, 60–62
  preconditioner setup function, 44, 70
  preconditioner solve function, 44, 69
  selection of, 32
  use in FCVODE, 86
CVSpbcg, 24, 29, 32
CVSPGMR linear solver
  Jacobian approximation used by, 44
  memory requirements, 60
  optional input, 44–46
  optional output, 60–62
  preconditioner setup function, 44, 70
  preconditioner solve function, 44, 69
  selection of, 32
  use in FCVODE, 86
CVSpgmr, 24, 29, 32
CVSPILS_ILL_INPUT, 32, 33, 45, 46, 72, 77
CVSPILS_LMEM_NULL, 45, 46, 60–62, 72, 77
CVSPILS_MEM_FAIL, 32, 33, 72, 77
CVSPILS_MEM_NULL, 32, 33, 45, 46, 60–62
CVSPILS_PMEM_NULL, 73, 77, 78
CVSPILS_SUCCESS, 32, 33, 45, 46, 60–62, 73
CVSpilsGetLastFlag, 62
CVSpilsGetNumConvFails, 60
CVSpilsGetNumJtimesEvals, 61
CVSpilsGetNumLinIters, 60
CVSpilsGetNumPrecEvals, 61
CVSpilsGetNumPrecSolves, 61
CVSpilsGetNumRhsEvals, 61
CVSpilsGetReturnFlagName, 62
CVSpilsGetWorkSpace, 60
CVSpilsJacTimesVecFn, 69
CVSpilsPrecSetupFn, 70
CVSpilsPrecSolveFn, 69
CVSpilsSetEpsLin, 46
CVSpilsSetGSType, 46
CVSpilsSetJacTimesFn, 45
CVSpilsSetMaxl, 46
CVSpilsSetPreconditioner, 45
CVSpilsSetPrecType, 45
CVSPTFQMR linear solver
  Jacobian approximation used by, 44
  memory requirements, 60
  optional input, 44–46
  optional output, 60
  preconditioner setup function, 44, 70
  preconditioner solve function, 44, 69
  selection of, 33
  use in FCVODE, 87
CVSptfqmr, 24, 29, 33
CVSUPERLUMT linear solver
  Jacobian approximation used by, 42
  matrix reordering algorithm specification, 43
  NVECTOR compatibility, 31
  optional input, 42–44
  optional output, 59
  selection of, 31
CVSuperLUMT, 24, 29, 31, 68
CVSuperLUMTSetOrdering, 44

data types
  Fortran, 79
DENSE generic linear solver
  functions
    large matrix, 167–168
    small matrix, 168–170

macros, 167
type DlsMat, 164–167
DENSE_COL, 66, 167
DENSE_ELEM, 66, 167
denseAddIdentity, 169
denseCopy, 169
denseGEQRF, 169
denseGETRF, 169
denseGETRS, 169
denseMatvec, 170
denseORMQR, 170
densePOTRF, 169
densePOTRS, 169
denseScale, 169
destroyArray, 169, 171
destroyMat, 168, 170
DlsMat, 66, 67, 164

eh_data, 64
error control
order selection, 12
step size selection, 12
error messages, 35
redirecting, 35
user-defined handler, 37, 64

FCVBAND, 84
FCVBANDSETJAC, 85
FCVBBDINIT, 94
FCVBBDOPT, 95
FCVBBDREINIT, 95
FCVBJAC, 84
FCVBPINIT, 93
FCVBPOPT, 93
FCVCOMMFN, 96
FCVDENSE, 83
FCVDENSESETJAC, 84
FCVDIAG, 83
FCVDJAC, 83
FCVDKY, 89
FCVEWT, 82
FCVEWTSET, 83
FCVFREE, 90
FCVFUN, 81
FCVGETERRWEIGHTS, 90
FCVGETESTLOCALERR, 92
FCVGLOCFN, 95
FCVJTIMES, 87, 96
FCVKLU, 85
FCVKLURENIT, 85
FCVMALLOC, 82
FCVMALLOC, 82
FCVODE, 88
FCVODE interface module
interface to the CVBANDPRE module, 93–94
interface to the CVBBDPRE module, 94–95
optional input and output, 90
rootfinding, 92
usage, 81–90
user-callable functions, 80–81
user-supplied functions, 81
FCVPSET, 88
FCVPSOL, 87
FCVREINIT, 89
FCVSETIIN, 90
FCVSETRIN, 90
FCVSPARSESETJAC, 86
FCVSPBCG, 86
FCVSPBCGREINIT, 89
FCVSPGMR, 86
FCVSPGMRREINIT, 89
FCVSPILSSETJAC, 87, 93, 95
FCVSPILSSETPREC, 87
FCVSPJAC, 86
FCVSPTFQMR, 87
FCVSPTFQMRREINIT, 89
FCVSUPERLUMT, 85
FGMRES method, 177

generic linear solvers
BAND, 164
DENSE, 164
KLU, 171
SLS, 171
SPBCG, 178
SPFGMR, 177
SPGMR, 176
SPTFQMR, 178
SUPERLUMT, 171
use in CVODE, 20
GMRES method, 32, 176
Gram-Schmidt procedure, 46

half-bandwidths, 30, 67–68, 72, 76
header files, 22, 71, 75
HNIL_WARNS, 90

INIT_STEP, 90
IOUT, 90, 91
itask, 24, **34**
iter, 26, 41

Jacobian approximation function
band
difference quotient, 42
use in FCVODE, 84
user-supplied, 42, 67–68
dense
difference quotient, 41

use in FCVODE, 83
user-supplied, 41, 65–66
diagonal
difference quotient, 32
Jacobian times vector
difference quotient, 44
use in FCVODE, 87
user-supplied, 45
Jacobian-vector product
user-supplied, 69
sparse
user-supplied, 42, 68–69

KLU sparse linear solver
type SlsMat, 172

linit, 160
lmm, 26, 63
LSODE, 1

MAX_CONVFAIL, 90
MAX_ERRFAIL, 90
MAX_NITERS, 90
MAX_NSTEPS, 90
MAX_ORD, 90
MAX_STEP, 90
maxl, 32, 33
maxord, 38, 63
memory requirements
CVBAND linear solver, 56
CVBANDPRE preconditioner, 73
CVBBDPRE preconditioner, 78
CVDENSE linear solver, 56
CVDIAG linear solver, 57
CVODE solver, 50
CVSPGMR linear solver, 60
MIN_STEP, 90
MODIFIED_GS, 46
MPI, 5

N_VCloneVectorArray, 98
N_VCloneVectorArray_OpenMP, 108
N_VCloneVectorArray_Parallel, 106
N_VCloneVectorArray_ParHyp, 112
N_VCloneVectorArray_Petsc, 114
N_VCloneVectorArray_Pthreads, 110
N_VCloneVectorArray_Serial, 103
N_VCloneVectorArrayEmpty, 98
N_VCloneVectorArrayEmpty_OpenMP, 108
N_VCloneVectorArrayEmpty_Parallel, 106
N_VCloneVectorArrayEmpty_ParHyp, 112
N_VCloneVectorArrayEmpty_Petsc, 114
N_VCloneVectorArrayEmpty_Pthreads, 110
N_VCloneVectorArrayEmpty_Serial, 103
N_VDestroyVectorArray, 98
N_VDestroyVectorArray_OpenMP, 108
N_VDestroyVectorArray_Parallel, 106
N_VDestroyVectorArray_ParHyp, 112
N_VDestroyVectorArray_Petsc, 114
N_VDestroyVectorArray_Pthreads, 111
N_VDestroyVectorArray_Serial, 103
N_Vector, 22, 97
N_VGetLength_OpenMP, 108
N_VGetLength_Parallel, 106
N_VGetLength_Pthreads, 111
N_VGetLength_Serial, 103
N_VGetLocalLength_Parallel, 106
N_VGetVector_ParHyp, 112
N_VGetVector_Petsc, 114
N_VMake_OpenMP, 108
N_VMake_Parallel, 106
N_VMake_ParHyp, 112
N_VMake_Petsc, 113
N_VMake_Pthreads, 110
N_VMake_Serial, 103
N_VNew_OpenMP, 108
N_VNew_Parallel, 105
N_VNew_Pthreads, 110
N_VNew_Serial, 103
N_VNewEmpty_OpenMP, 108
N_VNewEmpty_Parallel, 105
N_VNewEmpty_ParHyp, 112
N_VNewEmpty_Petsc, 113
N_VNewEmpty_Pthreads, 110
N_VNewEmpty_Serial, 103
N_VPrint_OpenMP, 108
N_VPrint_Parallel, 106
N_VPrint_ParHyp, 112
N_VPrint_Petsc, 114
N_VPrint_Pthreads, 111
N_VPrint_Serial, 104
newBandMat, 170
newDenseMat, 168
newIndexArray, 168, 170
newIntArray, 168, 171
newRealArray, 168, 171
NLCONV_COEF, 90
nonlinear system
definition, 9
Newton convergence test, 11
Newton iteration, 10–11
NV_COMM_P, 105
NV_CONTENT_OMP, 107
NV_CONTENT_P, 104
NV_CONTENT_PT, 109
NV_CONTENT_S, 102
NV_DATA_OMP, 107
NV_DATA_P, 105
NV_DATA_PT, 109

NV_DATA_S, 102
NV_GLOBLENGTH_P, 105
NV_Ith_OMP, 108
NV_Ith_P, 105
NV_Ith_PT, 110
NV_Ith_S, 103
NV_LENGTH_OMP, 107
NV_LENGTH_PT, 109
NV_LENGTH_S, 102
NV_LOCLENGTH_P, 105
NV_NUM_THREADS_OMP, 107
NV_NUM_THREADS_PT, 109
NV_OWN_DATA_OMP, 107
NV_OWN_DATA_P, 105
NV_OWN_DATA_PT, 109
NV_OWN_DATA_S, 102
NVECTOR module, 97

openMP, 5
optional input
band linear solver, 41–42
dense linear solver, 41–42
iterative linear solver, 44–46
rootfinding, 47
solver, 35–41
sparse linear solver, 42–44
optional output
band linear solver, 56–57
band-block-diagonal preconditioner, 77–78
banded preconditioner, 72–73
dense linear solver, 56–57
diagonal linear solver, 57–59
interpolated solution, 47
iterative linear solver, 60–62
solver, 48–55
sparse linear solver, 59
output mode, 13, 34

portability, 22
Fortran, 79
PREC_BOTH, 32, 33, 45
PREC_LEFT, 32, 33, 45
PREC_NONE, 32, 33, 45
PREC_RIGHT, 32, 33, 45
preconditioning
advice on, 13, 20
band-block diagonal, 73
banded, 71
setup and solve phases, 20
user-supplied, 44–45, 69, 70
pretype, 32, 33, 45
Pthreads, 5
PVODE, 1

RCONST, 22
realtype, 22
reinitialization, 62
right-hand side function, 63
Rootfinding, 14, 24, 33, 92
ROUT, 90, 91

SLS sparse linear solver
functions
small matrix, 175
SlsMat, 172
SM_COLS_B, 126
SM_COLS_D, 122
SM_COLUMN_B, 126
SM_COLUMN_D, 122
SM_COLUMN_ELEMENT_B, 126
SM_COLUMNS_B, 126
SM_COLUMNS_D, 122
SM_COLUMNS_S, 133
SM_CONTENT_B, 126
SM_CONTENT_D, 122
SM_CONTENT_DIAG, 129
SM_CONTENT_S, 133
SM_DATA_B, 126
SM_DATA_D, 122
SM_DATA_DIAG, 129
SM_DATA_S, 133
SM_ELEMENT_B, 126
SM_ELEMENT_D, 122
SM_INDEXPTRS_S, 133
SM_INDEXVALS_S, 133
SM_LBAND_B, 126
SM_LDATA_B, 126
SM_LDATA_D, 122
SM_LDIM_B, 126
SM_NNZ_S, 133
SM_NP_S, 133
SM_ROWS_B, 126
SM_ROWS_D, 122
SM_ROWS_S, 133
SM_SPARSETYPE_S, 133
SM_SUBAND_B, 126
SM_UBAND_B, 126
SMALL_REAL, 22
SparseAddIdentityMat, 175
SparseAddMat, 175
SparseCopyMat, 175
SparseDestroyMat, 175
SparseFromDenseMat, 175
SparseMatvec, 175
SparseNewMat, 175
SparsePrintMat, 175
SparseReallocMat, 175
SparseScaleMat, 175
sparsetype, 31

sparsetype=CSR_MAT, 31
SPBCG generic linear solver
description of, 178
functions, 178
SPFGMR generic linear solver
description of, 177
functions, 177
SPGMR generic linear solver
description of, 176
functions, 177
support functions, 177
SPTFQMR generic linear solver
description of, 178
functions, 178
STAB_LIM, 90
Stability limit detection, 13
step size bounds, 39
STOP_TIME, 90
SUNBandLinearSolver, 147
SUNBandMatrix, 127
SUNBandMatrix_Cols, 128
SUNBandMatrix_Column, 128
SUNBandMatrix_Columns, 127
SUNBandMatrix_Data, 128
SUNBandMatrix_LDim, 128
SUNBandMatrix_LowerBandwidth, 127
SUNBandMatrix_Print, 127
SUNBandMatrix_Rows, 127
SUNBandMatrix_StoredUpperBandwidth, 127
SUNBandMatrix_UpperBandwidth, 127
SUNDenseLinearSolver, 146
SUNDenseMatrix, 123
SUNDenseMatrix_Cols, 123
SUNDenseMatrix_Column, 123
SUNDenseMatrix_Columns, 123
SUNDenseMatrix_Data, 123
SUNDenseMatrix_LData, 123
SUNDenseMatrix_Print, 123
SUNDenseMatrix_Rows, 123
SUNDiagonalLinearSolver, 148
SUNDiagonalMatrix, 129
SUNDiagonalMatrix_Diag, 129
sundials_nvector.h, 22
sundials_types.h, 22
SUNKLU, 154
SUNKLUReInit, 154
SUNKLUSetOrdering, 154, 155
SUNLapackBand, 151
SUNLapackDense, 150
SUNLinearSolver, 139, 140
SUNLinearSolver module, 139
SUNLINEARSOLVER_CUSTOM, 141
SUNLINEARSOLVER_DIRECT, 141
SUNLINEARSOLVER_ITERATIVE, 141
SUNMatrix, 119
SUNMatrix module, 119
SUNSparseFromBandMatrix, 134
SUNSparseFromDenseMatrix, 134
SUNSparseMatrix, 133
SUNSparseMatrix_Columns, 135
SUNSparseMatrix_Data, 135
SUNSparseMatrix_IndexPointers, 135
SUNSparseMatrix_IndexValues, 135
SUNSparseMatrix_NNZ, 135
SUNSparseMatrix_NP, 135
SUNSparseMatrix_Print, 134
SUNSparseMatrix_Realloc, 134
SUNSparseMatrix_Rows, 134
SUNSparseMatrix_SparseType, 135
SUPERLUMT sparse linear solver
type SlsMat, 172

TFQMR method, 33, 46, 178
tolerances, 10, 27, 28, 65

UNIT_ROUNDOFF, 22
User main program
CVBANDPRE usage, 71
CVBBDPRE usage, 75
FCVBBD usage, 94
FCVBP usage, 93
FCVODE usage, 81
IVP solution, 23
user_data, 37, 64–68, 75

VODE, 1
VODPK, 1

weighted root-mean-square norm, 10