User Documentation for CVODES v2.8.2 (SUNDIALS v2.6.2)

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

July 30, 2015

UCRL-SM-208111

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

List of Figures

Chapter 1

Introduction

CVODES [31] is part of a software family called SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic equation Solvers [20]. This suite consists of CVODE, ARKODE, KINSOL and IDA, and variants of these with sensitivity analysis capabilities. CVODES is a solver for stiff and nonstiff initial value problems (IVPs) for systems of ordinary differential equation (ODEs). In addition to solving stiff and nonstiff ODE systems, CVODES has sensitivity analysis capabilities, using either the forward or the adjoint methods.

1.1 Historical background

FORTRAN solvers for ODE initial value problems are widespread and heavily used. Two solvers that were previously written at LLNL are VODE [3] and VODPK [5]. VODE is a general-purpose solver that includes methods for both stiff and nonstiff systems, and in the stiff case uses direct methods (full or banded) for the solution of the linear systems that arise at each implicit step. Externally, VODE is very similar to the well known solver LSODE [29]. VODPK is a variant of VODE that uses a preconditioned Krylov (iterative) method, namely [redacted]MRES, 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 were combined in the C-language package CVODE [10, 11].

At present, CVODES contains three Krylov methods that can be used in conjuction with Newton iteration: the [redacted]MRES ([redacted]eneralized Minimal RESidual) [30], Bi-C[redacted]Stab (Bi-Conjugate [redacted]radient Stabilized) [33], and TFQMR (Transpose-Free Quasi-Minimal Residual) linear iterative methods [15]. 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 CVODES, we recommend [redacted]MRES as the best overall choice. However, users are encouraged to compare all three, especially if encountering convergence failures with [redacted]MRES. Bi-C[redacted]FStab and TFQMR have an advantage in storage requirements, in that the number of workspace vectors they require is fixed, while that number for [redacted]MRES depends on the desired Krylov subspace size.

In the process of translating the VODE and VODPK algorithms into C, the overall CVODE organization has 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, thus 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 only a minimal impact on the rest of the solver, resulting in PVODE [7], the parallel variant of CVODE.

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

Development of CVODES was concurrent with a redesign of the vector operations module across the SUNDIALS suite. The key feature of the NVECTOR module is that it is written in terms of abstract vector operations with the actual vector functions attached by a particular implementation (such as serial or parallel) of NVECTOR. This allows writing the SUNDIALS solvers in a manner independent of the actual NVECTOR implementation (which can be user-supplied), as well as allowing more than one NVECTOR module to be linked into an executable file. SUNDIALS (and thus CVODES) is supplied with serial, MPI-parallel, and both openMP and Pthreads thread-parallel NVECTOR implementations.

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

1.2 Changes from previous versions

Changes in v2.8.0

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Changes in v2.7.0

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

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

A large number of minor errors have been fixed. Among these are the following: In CVSetTqBDF, the logic was changed to avoid a divide by zero. After the solver memory is created, it is set to zero before being filled. In each linear solver interface function, the linear solver memory is freed on an error return, and the **Free function now includes a line setting to NULL the main memory pointer to the linear solver memory. In the rootfinding functions cvRcheck1/cvRcheck2, when an exact zero is found, the array glo of g values at the left endpoint is adjusted, instead of shifting the t location tlo slightly. In the installation files, we modified the treatment of the macro SUNDIALS_USE_GENERIC_MATH, so that the parameter GENERIC_MATH_LIB is either defined (with no value) or not defined.

Changes in v2.6.0

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

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

The user interface has been further refined. Some of the API changes involve: (a) a reorganization of all linear solver modules into two families (besides the existing family of scaled preconditioned iterative linear solvers, the direct solvers, including the new Lapack-based ones, were also organized into a *direct* family); (b) maintaining a single pointer to user data, optionally specified through a Set-type function; (c) a general streamlining of the preconditioner modules distributed with the solver. Moreover, the prototypes of all functions related to integration of backward problems were modified to support the simultaneous integration of multiple problems. All backward problems defined by the user are internally managed through a linked list and identified in the user interface through a unique identifier.

Changes in v2.5.0

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

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

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

Changes in v2.4.0

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

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

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

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

Changes in v2.3.0

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

Changes in v2.2.0

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

Changes in v2.1.2

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

Changes in v2.1.1

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

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

Changes in v2.1.0

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

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

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

Installation of CVODES (and all of SUNDIALS) has been completely redesigned and is now based on a configure script.

1.3 Reading this user guide

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

There are different possible levels of usage of CVODES. The most casual user, with an IVP problem only, can get by with reading §2.1, then Chapter 4 through §4.5.5 only, and looking at examples in [32]. In addition, to solve a forward sensitivity problem the user should read §2.6, followed by Chapter 5 through §5.2.4 only, and look at examples in [32].

In a different direction, a more advanced user with an IVP problem may want to (a) use a package preconditioner (§4.8), (b) supply his/her own Jacobian or preconditioner routines (§4.6), (c) do multiple runs of problems of the same size (§4.5.9), (d) supply a new NVECTOR module (Chapter 7), or even (e) supply a different linear solver module (§3.2). An advanced user with a forward sensitivity problem may also want to (a) provide his/her own sensitivity equations right-hand side routine (§5.3), (b) perform multiple runs with the same number of sensitivity parameters (§5.2.1), or (c) extract additional diagnostic information (§5.2.4). A user with an adjoint sensitivity problem needs to understand the IVP solution approach at the desired level and also go through §2.7 for a short mathematical description of the adjoint approach, Chapter 6 for the usage of the adjoint module in CVODES, and the examples in [32].

The structure of this document is as follows:

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

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

- Chapter 4 is the main usage document for CVODES for simulation applications. It includes a complete description of the user interface for the integration of ODE initial value problems.

Readers that are not interested in using CVODES for sensitivity analysis can then skip the next two chapters.

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

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

- Chapter 7 gives a brief overview of the generic NVECTOR module shared amongst the various components of SUNDIALS, as well as details on the NVECTOR implementations provided with SUNDIALS: a serial implementation (§7.1), a distributed memory parallel implementation based on MPI (§7.2), and two thread-parallel implementations based on openMP (§7.3) and Pthreads (§7.4), respectively.

- Chapter 8 describes the specifications of linear solver modules as supplied by the user.

- Chapter 9 describes in detail the generic linear solvers shared by all SUNDIALS solvers.

- Finally, in the appendices, we provide detailed instructions for the installation of CVODES, within the structure of SUNDIALS (Appendix A), as well as a list of all the constants used for input to and output from CVODES functions (Appendix B).

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

Chapter 2

Mathematical Considerations

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

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

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

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

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

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

2.1 IVP solution

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

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

Here the y^n are computed approximations to $y(t_n)$, and $h_n = t_n - t_{n-1}$ is the step size. The user of CVODES must appropriately choose one of two multistep methods. For non-stiff problems, CVODES includes the Adams-Moulton formulas , characterized by $K_1 = 1$ and $K_2 = q$ above, where the order q varies between 1 and 12. For stiff problems, CVODES includes the Backward Differentiation Formulas (BDF) in so-called fixed-leading coefficient (FLC) form, given by $K_1 = q$ and $K_2 = 0$, with order q varying between 1 and 5. The coefficients are uniquely determined by the method type, its order, the recent history of the step sizes, and the normalization $\alpha_{n,0} = -1$. See [6] and [24].

For either choice of formula, the nonlinear system

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

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

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

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

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

in which

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

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

For the solution of the linear systems within the Newton corrections, CVODES provides several choices, including the option of an user-supplied linear solver module. The linear solver modules distributed with SUNDIALS are organized in 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, CVODES 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 [12, 1], or the thread-enabled SuperLU_MT sparse solver library [26, 13, 2] (serial or threaded vector modules only) [Note that users will need to download and install the KLU or SuperLU_MT packages independent of CVODES],
- a diagonal approximate Jacobian solver,
- SPGMR, a scaled preconditioned [redacted]MRES ([redacted]eneralized Minimal Residual method) solver without restarts,
- SPBCG, a scaled preconditioned Bi-C[redacted]Stab (Bi-Conjugate [redacted]radient 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 serial and threaded vector representations.

In the process of controlling errors at various levels, CVODES uses a weighted root-mean-square norm, denoted $\|\cdot\|_{\text{WRMS}}$, for all error-like quantities. The multiplicative weights used are based on the current solution and on the relative and absolute tolerances input by the user, namely

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

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

In the cases of a direct solver (dense, band, sparse, or diagonal), the iteration is a modified Newton iteration since 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 (for the direct solvers) 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 involve a reevaluation of J (in M) or of Jacobian data (in P) if 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 $\bar{\gamma}$ (γ at the last update) satisfies $|\gamma/\bar{\gamma} - 1| < 0.2$, or
- a convergence failure occurred that forced a reduction of the step size.

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

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

Now we use the estimate

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

Therefore the convergence (stopping) test is

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

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

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

With the direct dense and band methods, the Jacobian may be supplied by a user routine, or approximated by difference quotients, at the user's option. In the latter case, we use the usual approximation

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

The increments σ_j are given by

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

where U is the unit roundoff, σ_0 is a dimensionless value, and W_j is the error weight defined in (2.7). In the dense case, this scheme requires N evaluations of f, one for each column of J. In the band case, the columns of J are computed in groups by the Curtis-Powell-Reid algorithm, with the number of f evaluations equal to the bandwidth.

We note that with the sparse direct solvers, the Jacobian *must* be supplied by a user routine in 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

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

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

A critical part of CVODES, that makes 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

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

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

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

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

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

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

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

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

In addition to adjusting the step size to meet the local error test, CVODES periodically adjusts the order, with the goal of maximizing the step size. The integration starts out at order 1, but the order is varied 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 is made to the step size or order. At the current order q, selecting a new step size is done exactly as when the error test fails, giving a tentative step size ratio

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

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

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

and

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

The new order and step size are then set according to

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

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

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

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

2.2 Preconditioning

When using a Newton method to solve the nonlinear system (2.4), CVODES makes repeated use of a linear solver to solve linear systems of the form $Mx = -r$, where x is a correction vector and r is a residual vector. If this linear system solve is done with one of the scaled preconditioned iterative linear solvers, these solvers are rarely successful if used without preconditioning; it is generally necessary to precondition the system in order to obtain acceptable efficiency. A system $Ax = b$ can be preconditioned on the left, as $(P^{-1}A)x = P^{-1}b$; on the right, as $(AP^{-1})Px = b$; or on both sides, as $(P_L^{-1}AP_R^{-1})P_Rx = P_L^{-1}b$. The Krylov method is then applied to a system with the matrix $P^{-1}A$, or AP^{-1}, or $P_L^{-1}AP_R^{-1}$, instead of A. In order to improve the convergence of the Krylov iteration, the preconditioner matrix P, or the product P_LP_R in the last case, should in some sense approximate the system matrix A. Yet at the same time, in order to be cost-effective, the matrix P, or matrices P_L and P_R, should be reasonably efficient to evaluate and solve. Finding a good point in this tradeoff between rapid convergence and low cost can be very difficult. Good choices are often problem-dependent (for example, see [4] for an extensive study of preconditioners for reaction-transport systems).

The CVODES 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 CVODES are based on approximations to the system Jacobian, $J = \partial f/\partial y$. Since the Newton iteration matrix involved is $M = I - \gamma J$, any approximation $\bar{J}$ to J yields a matrix that is of potential use as a preconditioner, namely $P = I - \gamma \bar{J}$. Because the Krylov iteration occurs within a Newton iteration and further also within a time integration, and since each of these iterations has its own test for convergence, the preconditioner may use a very crude approximation, as long as it captures the dominant numerical feature(s) of the system. We have found that the combination of a preconditioner with the Newton-Krylov iteration, using even a fairly poor approximation to the Jacobian, can be surprisingly superior to using the same matrix without Krylov acceleration (i.e., a modified Newton iteration), as well as to using the Newton-Krylov method with no preconditioning.

2.3 BDF stability limit detection

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

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

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

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

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

The STALD algorithm attempts to detect, in a direct manner, the presence of a stability region boundary that is limiting the step sizes in the presence of a weakly damped oscillation [17]. The algorithm supplements (but differs greatly from) the existing algorithms in CVODES for choosing step size and order based on estimated local truncation errors. The STALD algorithm works directly with history data that is readily available in CVODES. If it concludes that the step size is in fact stability-limited, it dictates a reduction in the method order regardless of the outcome of the error-based algorithm. The STALD algorithm has been tested in combination with the VODE solver on linear advection-dominated advection-diffusion problems [18], where it works well. The implementation in CVODES has been successfully tested on linear and nonlinear advection-diffusion problems, among others.

This stability limit detection option adds some computational overhead to the CVODES solution. (In timing tests, these overhead costs have ranged from 2% to 7% of the total, depending on the size and complexity of the problem, with lower relative costs for larger problems.) Therefore, it should be activated only when there is reasonable expectation of modes in the user's system for which it is appropriate. In particular, if a CVODES solution with this option turned off appears to take an inordinately large number of steps for orders between 3 and 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 efficiency.

2.4 Rootfinding

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

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

The basic scheme used is to check for sign changes of any $g_i(t)$ over each time step taken, and then (when a sign change is found) to hone in on the root(s) with a modified secant method [16]. In addition, each time g is computed, CVODES checks to see if $g_i(t) = 0$ exactly, and if so it reports this as a root. However, if an exact zero of any g_i is found at a point t, CVODES computes g at $t+\delta$ for a small increment δ, slightly further in the direction of integration, and if any $g_i(t+\delta) = 0$ also, CVODES stops and reports an error. This way, each time CVODES takes a time step, it is guaranteed that the values of all g_i are nonzero at some past value of t, beyond which a search for roots is to be done.

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

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

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

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

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

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

2.5 Pure quadrature integration

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

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

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

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

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

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

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

once the new approximation y^n is available.

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

2.6 Forward sensitivity analysis

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

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

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

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

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

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

2.6.1 Forward sensitivity methods

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

- *Staggered Direct*

  In this approach [9], the nonlinear system (2.4) is first solved and, once an acceptable numerical solution is obtained, the sensitivity variables at the new step are found by directly solving (2.11) after the (BDF or Adams) discretization is used to eliminate $\dot{s}_i$. Although the system matrix of the above linear system is based on exactly the same information as the matrix M in (2.6), it must be updated and factored at every step of the integration, in contrast to an evalutaion of M which is updated only occasionally. For problems with many parameters (relative to the problem size), the staggered direct method can outperform the methods described below [25]. However, the computational cost associated with matrix updates and factorizations makes this method unattractive for problems with many more states than parameters (such as those arising from semidiscretization of PDEs) and is therefore not implemented in CVODES.

- *Simultaneous Corrector*

  In this method [27], the discretization is applied simultaneously to both the original equations (2.2) and the sensitivity systems (2.11) resulting in the following nonlinear system

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

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

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

  at each iteration, where

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

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

- *Staggered corrector*

  In this approach [14], as in the staggered direct method, the nonlinear system (2.4) is solved first using the Newton iteration (2.5). Then a separate Newton iteration is used to solve the sensitivity system (2.11):

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

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

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

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

2.6.2 Selection of the absolute tolerances for sensitivity variables

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

2.6.3 Evaluation of the sensitivity right-hand side

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

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

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

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

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

or simultaneously:

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

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

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

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

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

2.6.4 Quadratures depending on forward sensitivities

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

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

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

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

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

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

2.7 Adjoint sensitivity analysis

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

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

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

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

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

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

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

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

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

the gradient of G with respect to p is nothing but

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

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

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

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

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

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

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

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

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

2.7.1 Checkpointing scheme

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

However, especially for large-scale problems and long integration intervals, the number and size of the vectors y and $\dot{y}$ that would need to be stored make this approach computationally intractable.

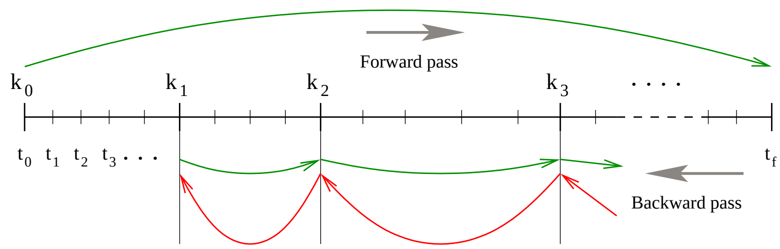


Figure 2.1: Illustration of the checkpointing algorithm for generation of the forward solution during the integration of the adjoint system.

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

This approach transfers the uncertainty in the number of integration steps in the forward integration phase to uncertainty in the final number of checkpoints. However, N_c is much smaller than the number of steps taken during the forward integration, and there is no major penalty for writing/reading the checkpoint data to/from a temporary file. Note that, at the end of the first forward integration stage, interpolation data are available from the last checkpoint to the end of the interval of integration. If no checkpoints are necessary (N_d is larger than the number of integration steps taken in the solution of (2.2)), the total cost of an adjoint sensitivity computation can be as low as one forward plus one backward integration. In addition, CVODES provides the capability of reusing a set of checkpoints for multiple backward integrations, thus allowing for efficient computation of gradients of several functionals (2.16).

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

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

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

2.8 Second-order sensitivity analysis

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

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

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

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

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

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

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

where λ, μ, and s are solutions of

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

In the above equation, $s = y_p u$ is a linear combination of the columns of the sensitivity matrix y_p. The *forward-over-adjoint* approach hinges crucially on the fact that s can be computed at the cost of a forward sensitivity analysis with respect to a single parameter (the last ODE problem above) which is possible due to the linearity of the forward sensitivity equations (2.11).

Therefore, the cost of computing the Hessian-vector product is roughly that of two forward and two backward integrations of a system of ODEs of size N. For more details, including the corresponding formulas for a pointwise model functional output, see [28].

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

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

[2]For the sake of simplifity in presentation, we do not include explicit dependencies of g on time t or parameters p. Moreover, we only consider the case in which the dependency of the original ODE (2.2) on the parameters p is through its initial conditions only. For details on the derivation in the general case, see [28].

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:

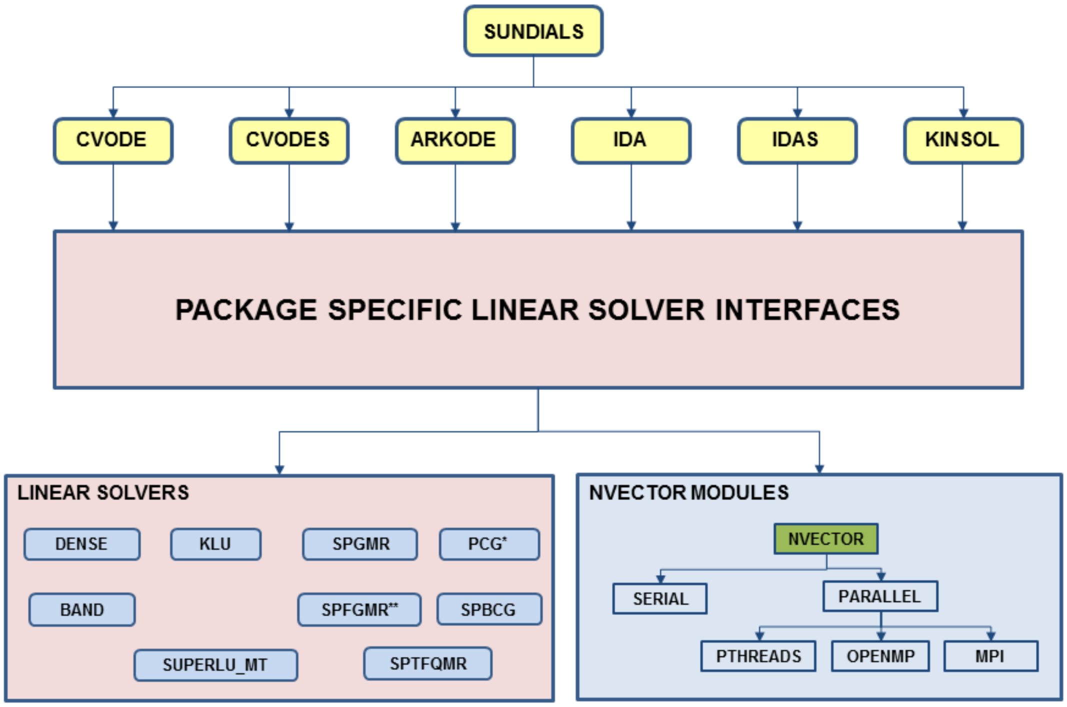
- CVODE, a solver for stiff and nonstiff ODE systems $dy/dt = f(t, y)$ based on Adams and BDF methods;
- CVODES, a solver for stiff and nonstiff ODE systems with sensitivity analysis capabilities;
- ARKODE, a solver for ODE systems $M dy/dt = f(t, y)$ based on additive Runge-Kutta methods;
- IDA, a solver 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 CVODES organization

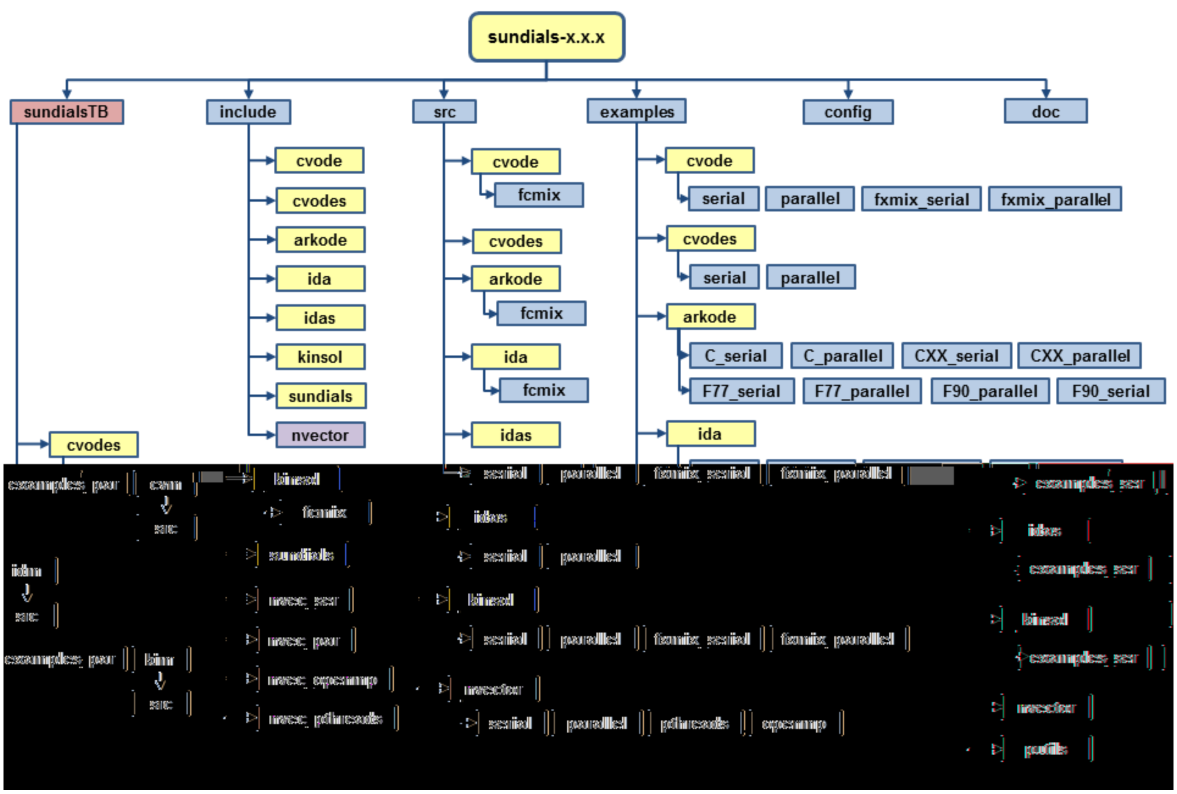
The CVODES package is written in ANSI C. The following summarizes the basic structure of the package, although knowledge of this structure is not necessary for its use.

The overall organization of the CVODES package is shown in Figure 3.2. The basic elements of the structure are a module for the basic integration algorithm (including forward sensitivity analysis), a module for adjoint sensitivity analysis, and a set of modules for the solution of linear systems that arise in the case of a stiff system. The central integration module, implemented in the files `cvodes.h`, `cvodes_impl.h`, and `cvodes.c`, deals with the evaluation of integration coefficients, the functional or Newton iteration process, estimation of local error, selection of step size 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.

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



(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

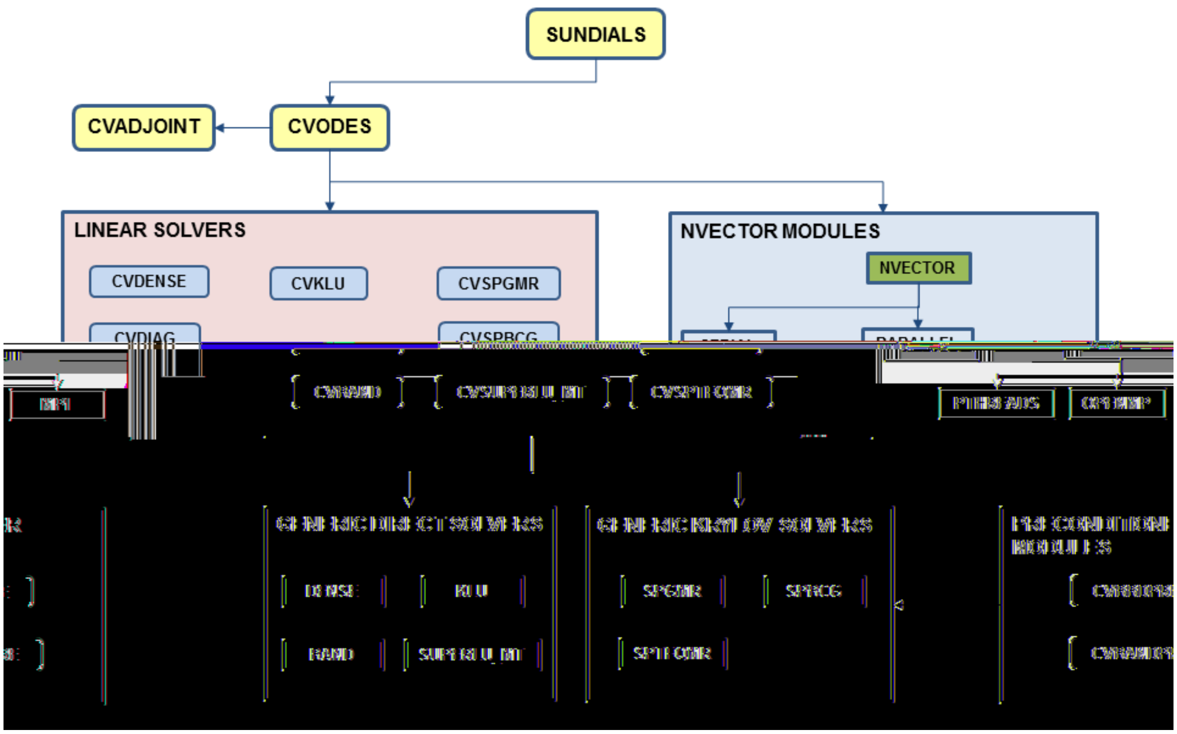


Figure 3.2: Overall structure of the CVODES package. Modules specific to CVODES are distinguished by rounded boxes, while generic solver and auxiliary modules are in rectangular boxes. Note that the direct linear solvers using Lapack implementations are not explicitly represented. Note also that the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

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

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

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

- CVDENSE: LU factorization and backsolving with dense matrices (using either an internal implementation or Blas/Lapack);
- 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 [12, 1] (KLU to be downloaded and compiled by user independent of CVODES);
- CVSUPERLUMT: LU factorization and backsolving with compressed-sparse-column (CSC) matrices using the threaded SuperLU_MT linear solver library [26, 13, 2] (SuperLU_MT to be downloaded and compiled by user independent of CVODES).

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

- CVSPGMR: scaled preconditioned ●●●2y meh0.42149811(d);

oduldistr di0.4414975(g)-414.216(cvodes)0.423947(i)0.441101(t)

797718 431.452

preconditioners is done only periodically during the integration, and only as required to achieve convergence. The call list within the central CVODES module for 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 CVODES 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 CVODES package elsewhere.

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

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

Chapter 4

Using CVODES for IVP Solution

This chapter is concerned with the use of CVODES for the solution of initial value problems (IVPs). The following sections treat the header files and the layout of the user's main program, and provide descriptions of the CVODES user-callable functions and user-supplied functions. This usage is essentially equivalent to using CVODE [22].

The sample programs described in the companion document [32] may also be helpful. Those codes may be used as templates (with the removal of some lines used in testing) and are included in the CVODES package.

The user should be aware that not all 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 CVDENSE and CVBAND modules (using either the internal implementation or Lapack), as well as the CVKLU, CVSUPERLUMT and CVBANDPRE modules can only be used with NVECTOR_SERIAL, NVECTOR_OPENMP and NVECTOR_PTHREADS. 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.

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

4.1 Access to library and header files

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

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

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

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

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

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).

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

4.2 Data Types

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

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

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

The calling program must also include an NVECTOR implementation header file (see Chapter 7 for details). For the NVECTOR implementations that are included in the CVODES package, the corresponding header files are:

- `nvector_serial.h`, which defines the serial implementation NVECTOR_SERIAL;
- `nvector_parallel.h`, which defines the parallel (MPI) implementation, NVECTOR_PARALLEL.
- `nvector_openmp.h`, which defines the shared memory parallel openMP implementation,
- `nvector_pthreads.h`, which defines the shared memory parallel Pthreads implementation.

Note that both these files in turn include 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 CVODES are:

- `cvodes_dense.h`, which is used with the dense direct linear solver;
- `cvodes_band.h`, which is used with the band direct linear solver;
- `cvodes_lapack.h`, which is used with Lapack implementations of dense or band direct linear solvers;
- `cvodes_diag.h`, which is used with the diagonal linear solver;
- `cvodes_klu.h`, which is used with the KLU sparse direct linear solver;
- `cvodes_superlumt.h`, which is used with the SuperLU_MT threaded sparse direct linear solver;
- `cvodes_spgmr.h`, which is used with the scaled, preconditioned GMRES Krylov linear solver SPGMR;
- `cvodes_spbcgs.h`, which is used with the scaled, preconditioned Bi-CGStab Krylov linear solver SPBCG;
- `cvodes_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 `cvodes_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 `cvodes_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 `cvodes_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 `cvsDiurnal_kry_p` example (see [32]), 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. Some steps are independent of the NVECTOR implementation used; where this is not the case, usage specifications are given for the two implementations provided with CVODES: steps marked [**P**] correspond to NVECTOR_PARALLEL, steps marked [**O**] correspond to NVECTOR_OPENMP, steps marked [**T**] correspond to NVECTOR_PTHREADS, while steps marked [**S**] correspond to NVECTOR_SERIAL.

1. **[P] Initialize MPI**

   Call `MPI_Init(&argc, &argv)` to initialize MPI if used by the user's program. Here `argc` and `argv` are the command line argument counter and array received by `main`, respectively.

2. **Set problem dimensions**

   [**S**], [**O**], [**T**] Set `N`, the problem size N.

   [**O**], [**T**] Set `num_threads`, the number of threads to use within the threaded vector functions.

   [**P**] Set `Nlocal`, the local vector length (the sub-vector length for this process); `N`, the global vector length (the problem size N, and the sum of all the values of `Nlocal`); and the active set of processes.

   Note: The variables `N` and `Nlocal` should be of type `long int`. The variable `num_threads` should be of type `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. If a `realtype` array `ydata` containing the initial values of y already exists, then make the call:

   [**S**] `y0 = N_VMake_Serial(N, ydata);`

   [**O**] `y0 = N_VMake_OpenMP(N, num_threads, ydata);`

   [**T**] `y0 = N_VMake_Pthreads(N, num_threads, ydata);`

   [**P**] `y0 = N_VMake_Parallel(comm, Nlocal, N, ydata);`

   Otherwise, make the call:

   [**S**] `y0 = N_VNew_Serial(N);`

   [**O**] `y0 = N_VNew_OpenMP(N, num_threads);`

   [**T**] `y0 = N_VNew_Pthreads(N, num_threads);`

   [**P**] `y0 = N_VNew_Parallel(comm, Nlocal, N);`

   and load initial values into the structure defined by:

   [**S**] `NV_DATA_S(y0)`

   [**O**] `NV_DATA_OMP(y0)`

   [**T**] `NV_DATA_PT(y0)`

   [**P**] `NV_DATA_P(y0)`

   Here `comm` is the MPI communicator, set in one of two ways: If a proper subset of active processes is to be used, `comm` must be set by suitable MPI calls. Otherwise, to specify that all processes are to be used, `comm` must be `MPI_COMM_WORLD`.

4. **Create CVODES object**

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

5. **Initialize CVODES solver**

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

6. **Specify integration tolerances**

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

7. **Set optional inputs**

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

8. **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):

   [**S**], [**O**], [**T**] `ier = CVDense(...);`

   [**S**], [**O**], [**T**] `ier = CVBand(...);`

   [**S**], [**O**], [**T**] `ier = CVLapackDense(...);`

   [**S**], [**O**], [**T**] `ier = CVLapackBand(...);`

   `ier = CVDiag(...);`

   [**S**], [**O**], [**T**] `ier = CVKLU(...);`

   [**S**], [**O**], [**T**] `ier = CVSuperLUMT(...);`

   `ier = CVSpgmr(...);`

   `ier = CVSpbcg(...);`

   `ier = CVSptfqmr(...);`

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 `y` (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` by calling the destructor function defined by the NVECTOR implementation:

    [**S**] `N_VDestroy_Serial(y);`

    [**O**] `N_VDestroy_OpenMP(y);`

    [**T**] `N_VDestroy_Pthreads(y);`

    [**P**] `N_VDestroy_Parallel(y);`

14. **Free solver memory**

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

15. **[P] Finalize MPI**

    Call `MPI_Finalize()` to terminate MPI.

4.5 User-callable functions

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

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

4.5.1 CVODES initialization and deallocation functions

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

`CVodeCreate`

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

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

Arguments `lmm` (`int`) specifies the linear multistep method and may be one of two possible values: `CV_ADAMS` or `CV_BDF`.

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

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

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

`CVodeInit`

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

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

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

`f` (`CVRhsFn`) is the C function which computes the right-hand side function f in the ODE. This function has the form `f(t, y, ydot, user_data)` (for full details see §4.6.1).

`t0` (`realtype`) is the initial value of t.

`y0` (`N_Vector`) is the initial value of y.

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

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

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

`CV_MEM_FAIL` A memory allocation request has failed.

`CV_ILL_INPUT` An input argument to `CVodeInit` has an illegal value.

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

`CVodeFree`

Call `CVodeFree(&cvode_mem);`

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

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

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

4.5.2 CVODES tolerance specification functions

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

`CVodeSStolerances`

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

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

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

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

`CVodeSVtolerances`

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

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

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

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

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

`CVodeWFtolerances`

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

Description The function `CVodeWFtolerances` specifies a user-supplied function `efun` that sets the multiplicative error weights W_i for use in the weighted RMS norm, which are normally defined by Eq. (2.7).

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

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

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

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

`CV_NO_MALLOC` The allocation function `CVodeInit` has not been called.

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

(1) The scalar relative tolerance `reltol` is to be set to control relative errors. So `reltol` $= 10^{-4}$ means that errors are controlled to .01%. We do not recommend using `reltol` larger than 10^{-3}. On the other hand, `reltol` should not be so small that it is comparable to the unit roundoff of the machine arithmetic (generally around 1.0E-15).

(2) The absolute tolerances `abstol` (whether scalar or vector) need to be set to control absolute errors when any components of the solution vector `y` may be so small that pure relative error control is meaningless. For example, if `y[i]` starts at some nonzero value, but in time decays to zero, then pure relative error control on `y[i]` makes no sense (and is overly costly) after `y[i]` is below some noise level. Then `abstol` (if scalar) or `abstol[i]` (if a vector) needs to be set to that noise level. If the different components have different noise levels, then `abstol` should be a vector. See the example `cvsRoberts_dns` in the CVODES package, and the discussion of it in the CVODE Examples document [21]. 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 CVODES, with magnitude comparable to `abstol` or less, is equivalent to zero as far as the computation is concerned.

(3) The user's right-hand side routine `f` should never change a negative value in the solution vector `y` to a non-negative value, as a "solution" to this problem. This can cause instability. If the `f` routine cannot tolerate a zero or negative value (e.g. because there is a square root or log of it), then the

offending value should be changed to zero or a tiny positive number in a temporary variable (not in the input y vector) for the purposes of computing $f(t, y)$.

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

4.5.3 Linear solver specification functions

As previously explained, Newton iteration requires the solution of linear systems of the form (2.5). There are eight CVODES 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 CVODES 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 $P_1 P_2$ approximates the Newton matrix $M = I - \gamma J$ of (2.6).

To specify a CVODES 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`, `CVDiag`, `CVKLU`, `CVSuperLUMT`, `CVSpgmr`, `CVSpbcg`, or `CVSptfqmr`, as documented below. The first argument passed to these functions is the CVODES memory pointer returned by `CVodeCreate`. A call to one of these functions links the main CVODES 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 CVODES 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 9.

`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 `cvodes_dense.h` header file.

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

`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 `cvodes_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.

`CVDiag`

Call `flag = CVDiag(cvode_mem);`

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

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

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

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

`CVDIAG_SUCCESS` The CVDIAG initialization was successful.

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

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

`CVDIAG_MEM_FAIL` A memory allocation request failed.

Notes The CVDIAG solver is the simplest of all of the current CVODES linear 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.

`CVKLU`

Call `flag = CVKLU(cvode_mem, N, NNZ);`

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 `cvodes_klu.h` header file.

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

`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 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 `cvodes_superlumt.h` header file.

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

`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 `cvodes_spgmr.h` header file.

Arguments
`cvode_mem` (`void *`) pointer to the CVODES 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 (2.5).

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 `cvodes_spbcgs.h` header file.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES 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 (2.5).

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 `cvodes_sptfqmr.h` header file.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES 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 (2.5).

4.5.4 Rootfinding initialization function

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

CVodeRootInit

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

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

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

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

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

CV_SUCCESS	The call to CVodeRootInit was successful.
CV_MEM_NULL	The cvode_mem argument was NULL.
CV_MEM_FAIL	A memory allocation failed.
CV_ILL_INPUT	The function g is NULL, but nrtfn > 0.

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

4.5.5 CVODES solver function

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

CVode

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

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

Arguments

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

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

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

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

CV_SUCCESS	CVode succeeded and no roots were found.
CV_TSTOP_RETURN	CVode succeeded by reaching the stopping point specified through the optional input function CVodeSetStopTime (see §4.5.6.1).
CV_ROOT_RETURN	CVode succeeded and found one or more roots. In this case, tret is the location of the root. If nrtfn > 1, call CVodeGetRootInfo to see which g_i were found to have a root.
CV_MEM_NULL	The cvode_mem argument was NULL.
CV_NO_MALLOC	The CVODES memory was not allocated by a call to CVodeInit.
CV_ILL_INPUT	One of the inputs to CVode was illegal, or some other input to the solver was either illegal or missing. The latter category includes the following situations: (a) The tolerances have not been set. (b) A component of the error weight vector became zero during internal time-stepping. (c) The linear solver initialization function (called by

the user after calling `CVodeCreate`) failed to set the linear solver-specific `lsolve` field in `cvode_mem`. (d) A root of one of the root functions was found both at a point t and also very near t. In any case, the user should see the error message for details.

`CV_TOO_CLOSE` The initial time t_0 and the final time t_{out} are too close to each other and the user did not specify an initial step size.

`CV_TOO_MUCH_WORK` The solver took `mxstep` internal steps but still could not reach `tout`. The default value for `mxstep` is `MXSTEP_DEFAULT = 500`.

`CV_TOO_MUCH_ACC` The solver could not satisfy the accuracy demanded by the user for some internal step.

`CV_ERR_FAILURE` Either error test failures occurred too many times (`MXNEF = 7`) during one internal time step, or with $|h| = h_{min}$.

`CV_CONV_FAILURE` Either convergence test failures occurred too many times (`MXNCF = 10`) during one internal time step, or with $|h| = h_{min}$.

`CV_LINIT_FAIL` The linear solver's initialization function failed.

`CV_LSETUP_FAIL` The linear solver's setup function failed in an unrecoverable manner.

`CV_LSOLVE_FAIL` The linear solver's solve function failed in an unrecoverable manner.

`CV_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 CVODES solver. CVODES provides functions that can be used to change these optional input parameters from their default values. Table 4.1 lists all optional input functions in CVODES which are then described in detail in the remainder of this section, begining with those for the main CVODES solver and continuing with those for the linear solver modules. Note that the diagonal linear solver module has no optional inputs. For the most casual use of CVODES, the reader can skip to §4.6.

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

Table 4.1: Optional inputs for CVODES, CVDLS, CVSLS, and CVSPILS

Optional input	**Function name**	**Default**
	CVODES main solver	
Pointer to an error file	`CVodeSetErrFile`	`stderr`
Error handler function	`CVodeSetErrHandlerFn`	internal fn.
User data	`CVodeSetUserData`	`NULL`
Maximum order for BDF method	`CVodeSetMaxOrd`	5
Maximum order for Adams method	`CVodeSetMaxOrd`	12
Maximum no. of internal steps before $t_{\rm out}$	`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 [redacted]ram-Schmidt orthogonalization$^{(a)}$	`CVSpilsSetGSType`	classical [redacted]S
Maximum Krylov subspace size$^{(b)}$	`CVSpilsSetMaxl`	5

$^{(a)}$ Only for CVSPGMR

$^{(b)}$ Only for CVSPBCG and CVSPTFQMR

4.5.6.1 Main solver optional input functions

The calls listed here can be executed in any order. However, if either of the functions `CVodeSetErrFile` or `CVodeSetErrHandlerFn` is to be called, that call should be first, in order to take effect for any later error message.

`CVodeSetErrFile`

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

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

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

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

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

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

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

`CVodeSetErrHandlerFn`

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

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

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

Return value The return value `flag` (of type `int`) is one of
- `CV_SUCCESS` The function `ehfun` and data pointer `eh_data` have been successfully set.
- `CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.

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

`CVodeSetUserData`

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

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

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

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

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

If `user_data` is needed in user 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 CVODES memory block.
- `maxord` (`int`) value of the maximum method order. This must be positive.

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

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

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

CVodeSetMaxNumSteps

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

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

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

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

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

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

CVodeSetMaxHnilWarns

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

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

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

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

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

`CVodeSetStabLimDet`

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

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `stldet` (`booleantype`) flag controlling stability limit detection (`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, `hmax` (`realtype`) maximum absolute value of the step size (≥ 0.0).

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

`CV_SUCCESS` The optional value has been successfully set.

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

`CV_ILL_INPUT` Either `hmax` is nonpositive or it is smaller than the minimum allowable step size.

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

`CVodeSetStopTime`

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

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

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

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

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

`CV_SUCCESS` The optional value has been successfully set.

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

`CV_ILL_INPUT` The value of `tstop` is not beyond the current t value, t_n.

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

`CVodeSetMaxErrTestFails`

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

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

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

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

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

`CV_SUCCESS` The optional value has been successfully set.

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

Notes The default value is 7.

`CVodeSetMaxNonlinIters`

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

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

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

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

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

`CV_SUCCESS` The optional value has been successfully set.

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

Notes The default value is 3.

`CVodeSetMaxConvFails`

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

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

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

`maxncf` (`int`) maximum number of allowable nonlinear solver convergence failures per step (> 0).

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

`CV_SUCCESS` The optional value has been successfully set.

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

Notes The default value is 10.

`CVodeSetNonlinConvCoef`

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

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

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

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

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

`CV_SUCCESS` The optional value has been successfully set.

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

Notes The default value is 0.1.

`CVodeSetIterType`

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

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

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

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

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

`CV_SUCCESS` The optional value has been successfully set.

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

`CV_ILL_INPUT` The `iter` value passed is neither `CV_NEWTON` nor `CV_FUNCTIONAL`.

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

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 CVODES 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 CVODES 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 CVODES 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 CVODES 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 CVODES 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 CVODES 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 CVODES 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 CVODES 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 CVODES 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 [redacted]ram-Schmidt orthogonalization to be used with the CVSPGMR solver (one of the enumeration constants `MODIFIED_GS` or `CLASSICAL_GS`). These correspond to using modified [redacted]ram-Schmidt and classical [redacted]ram-Schmidt, respectively.

Arguments —
`cvode_mem` (`void *`) pointer to the CVODES memory block.
`gstype` (`int`) type of [redacted]ram-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 CVODES memory block.
`eplifac` (`realtype`) linear convergence safety factor (≥ 0.0).

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

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

Notes — The default value is 0.05.

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-C[redacted]Stab or TFQMR methods.

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

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

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

`CVodeSetNoInactiveRootWarn`

Call: `flag = CVodeSetNoInactiveRootWarn(cvode_mem);`

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

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

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

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

4.5.7 Interpolated output function

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

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

`CVodeGetDky`

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

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

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

Return value The return value `flag` (of type `int`) is one of
- `CV_SUCCESS` `CVodeGetDky` succeeded.
- `CV_BAD_K` `k` is not in the range $0, 1, \ldots, q_u$.
- `CV_BAD_T` `t` is not in the interval $[t_n - h_u, t_n]$.
- `CV_BAD_DKY` The `dky` argument was `NULL`.
- `CV_MEM_NULL` The `cvode_mem` argument was `NULL`.

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

4.5.8 Optional output functions

CVODES provides an extensive set of functions that can be used to obtain solver performance information. Table 4.2 lists all optional output functions in CVODES, which are then described in detail in the remainder of this section.

Some of the optional outputs, especially the various counters, can be very useful in determining how successful the CVODES solver is in doing its job. For example, the counters `nsteps` and `nfevals` provide a rough measure of the overall cost of a given run, and can be compared among runs with differing input options to suggest which set of options is most efficient. The ratio `nniters/nsteps` measures the performance of the Newton iteration in solving the nonlinear systems at each time step; typical values for this range from 1.1 to 1.8. The ratio `njevals/nniters` (in the case of a direct linear solver), and the ratio `npevals/nniters` (in the case of an iterative linear solver) measure the overall degree of nonlinearity in these systems, and also the quality of the approximate Jacobian or preconditioner being used. Thus, for example, `njevals/nniters` can indicate if a user-supplied Jacobian is inaccurate, if this ratio is larger than for the case of the corresponding internal Jacobian. The ratio `nliters/nniters` measures the performance of the Krylov iterative linear solver, and thus (indirectly) the quality of the preconditioner.

4.5.8.1 Main solver optional output functions

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

Table 4.2: Optional outputs from CVODES, CVDLS, CVDIAG, CVSLS, and CVSPILS

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

`CVodeGetWorkSpace`

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

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

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

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

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

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

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` $= 50$

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

`CVodeGetNumSteps`

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

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `nsteps` (`long int`) number of steps taken by CVODES.

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

`CVodeGetNumRhsEvals`

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

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

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

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

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

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

`CVodeGetNumLinSolvSetups`

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

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

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

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

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

`CVodeGetNumErrTestFails`

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

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

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

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

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

`CVodeGetLastOrder`

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

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

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

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

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

`CVodeGetCurrentOrder`

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

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

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

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

`CVodeGetLastStep`

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

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

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

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

`CVodeGetCurrentStep`

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

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

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

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

`CVodeGetActualInitStep`

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

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

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

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

Notes Even if the value of the initial integration step size was specified by the user through a call to `CVodeSetInitStep`, this value might have been changed by CVODES to ensure that the step size is within the prescribed bounds ($h_{\min} \le h_0 \le h_{\max}$), or to satisfy the local error test condition.

`CVodeGetCurrentTime`

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

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

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

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

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

`CVodeGetNumStabLimOrderReds`

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

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

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

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

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

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

`CVodeGetTolScaleFactor`

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

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

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

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

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

`CVodeGetErrWeights`

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

Description The function `CVodeGetErrWeights` returns the solution error weights at the current time. These are the reciprocals of the W_i given by (2.7).

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

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

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

Notes The user must allocate memory for `eweight`.

`CVodeGetEstLocalErrors`

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

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

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

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

Notes The user must allocate memory for `ele`.

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

The `ele` vector, togther with the `eweight` vector from `CVodeGetErrWeights`, can be used to determine how the various components of the system contributed to the estimated local error test. Specifically, that error test uses the RMS norm of a vector whose components are the products of the components of these two vectors. Thus, for example, if there were recent error test failures, the components causing the failures are those with largest values for the products, denoted loosely as `eweight[i]*ele[i]`.

`CVodeGetIntegratorStats`

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

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

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

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

`CVodeGetNumNonlinSolvIters`

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

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

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

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

CV_SUCCESS The optional output values have been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

`CVodeGetNumNonlinSolvConvFails`

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

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

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

nncfails (long int) number of nonlinear convergence failures.

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

`CVodeGetNonlinSolvStats`

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

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

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

nniters (long int) number of nonlinear iterations performed.

nncfails (long int) number of nonlinear convergence failures.

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

CV_SUCCESS The optional output value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

`CVodeGetReturnFlagName`

Call `name = CVodeGetReturnFlagName(flag);`

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

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

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

4.5.8.2 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 CVODES memory block.

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

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

CV_SUCCESS The optional output values have been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

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

The user must allocate memory for the vector rootsfound.

CVodeGetNumGEvals

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

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

Arguments:
- cvode_mem (void *) pointer to the CVODES memory block.
- ngevals (long int) number of calls made to the user's function g thus far.

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

4.5.8.3 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 CVODES memory block.
- lenrwLS (long int) the number of realtype values in the CVDLS workspace.
- leniwLS (long int) the number of integer values in the CVDLS workspace.

Return value: The return value flag (of type int) is one of
- CVDLS_SUCCESS The optional output values have been successfully set.
- CVDLS_MEM_NULL The cvode_mem pointer is NULL.
- CVDLS_LMEM_NULL The CVDLS linear solver has not been initialized.

Notes: 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 $(2\,\texttt{mupper}+3\,\texttt{mlower}+2)\,N$ 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 CVODES memory block.
- njevals (long int) the number of calls to the Jacobian function.

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

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

`CVDlsGetNumRhsEvals`

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

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

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

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

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

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

Notes The value `nfevalsLS` is incremented only if 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 CVODES memory block.

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

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

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

Notes If the 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 CVODES memory block.
- lenrwLS (long int) the number of realtype values in the CVDIAG workspace.
- leniwLS (long int) the number of integer values in the CVDIAG workspace.

Return value The return value flag (of type int) is one of
- CVDIAG_SUCCESS The optional output valus have been successfully set.
- CVDIAG_MEM_NULL The cvode_mem pointer is NULL.
- CVDIAG_LMEM_NULL The CVDIAG linear solver has not been initialized.

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

CVDiagGetNumRhsEvals

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

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

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

Return value The return value flag (of type int) is one of
- CVDIAG_SUCCESS The optional output value has been successfully set.
- CVDIAG_MEM_NULL The cvode_mem pointer is NULL.
- CVDIAG_LMEM_NULL The CVDIAG linear solver has not been initialized.

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

CVDiagGetLastFlag

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

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

Arguments
- cvode_mem (void *) pointer to the CVODES memory block.
- lsflag (long int) the value of the last return flag from a CVDIAG function.

Return value The return value flag (of type int) is one of
- CVDIAG_SUCCESS The optional output value has been successfully set.
- CVDIAG_MEM_NULL The cvode_mem pointer is NULL.
- CVDIAG_LMEM_NULL The CVDIAG linear solver has not been initialized.

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

`CVDiagGetReturnFlagName`

Call `name = CVDiagGetReturnFlagName(lsflag);`

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

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

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

4.5.8.5 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 CVODES memory block.
njevals (long int) the number of calls to the Jacobian function.

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

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 CVODES 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 CVODES memory block.
- `lenrwLS` (`long int`) the number of `realtype` values in the CVSPILS workspace.
- `leniwLS` (`long int`) the number of integer values in the CVSPILS workspace.

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

Notes In terms of the problem size N and maximum subspace size `maxl`, the actual size of the real workspace is roughly:
$(\texttt{maxl}+5) * N + \texttt{maxl} * (\texttt{maxl}+4) + 1$ `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 CVODES memory block.
- `nliters` (`long int`) the current number of linear iterations.

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

`CVSpilsGetNumConvFails`

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

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

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

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

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

CVSpilsGetNumPrecEvals

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

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

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

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

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

CVSpilsGetNumPrecSolves

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

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

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

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

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

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 CVODES memory block.
`njvevals` (`long int`) the current number of calls to `jtimes`.

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

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

CVSpilsGetNumRhsEvals

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

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

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

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

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

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

`CVSpilsGetLastFlag`

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

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

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

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

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

Notes If the CVSPILS setup function failed (CVode returned CV_LSETUP_FAIL), lsflag will be 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 CVODES reinitialization function

The function CVodeReInit reinitializes the main CVODES solver for the solution of a 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.

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

`CVodeReInit`

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

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

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

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

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

4.6 User-supplied functions

The user-supplied functions consist of one function defining the ODE, (optionally) a function that handles error and warning messages, (optionally) a function that provides the error weight vector, (optionally) 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 CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CV_RHSFUNC_FAIL` is returned).

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

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

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

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

4.6.2 Error message handler function

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

`CVErrHandlerFn`

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

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

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

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

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

4.6.3 Error weight function

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

`CVEwtFn`

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

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

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

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

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

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

4.6.4 Rootfinding function

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

`CVRootFn`

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

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

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

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

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

4.6.5 Jacobian information (direct method 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 CVODES 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 §9.1.3.

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 CVODES 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` $\leq$ `m-n` $\leq$ `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 §9.1.4.

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 CVODES 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, CVODES 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 §9.2.

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 Integration of pure quadrature equations

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

1. **[P] Initialize MPI**
2. **Set problem dimensions**

   [**S**] Set `N` to the problem size N (excluding quadrature variables), and `Nq` to the number of quadrature variables.

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

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

   Typically, the quadrature variables should be initialized to 0.

10. **Initialize quadrature integration**

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

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

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

12. **Advance solution in time**

13. **Extract quadrature variables**

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

14. **Get optional outputs**

15. **Get quadrature optional outputs**

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

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

17. **Free solver memory**

18. **[P] Finalize MPI**

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

4.7.1 Quadrature initialization and deallocation functions

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

`CVodeQuadInit`

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

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

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

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

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

CV_REPTD_QRHSFUNC_ERR Convergence test failures occurred too many times due to repeated recoverable errors in the quadrature right-hand side function. This value will also be returned if the quadrature right-hand side function had repeated recoverable errors during the estimation of an initial step size (assuming the quadrature variables are included in the error tests).

CV_UNREC_RHSFUNC_ERR The quadrature right-hand function had a recoverable error, but no recovery was possible. This failure mode is rare, as it can occur only if the quadrature right-hand side function fails recoverably after an error test failed while at order one.

4.7.3 Quadrature extraction functions

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

`CVodeGetQuad`

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

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

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

Return value The return value `flag` of `CVodeGetQuad` is one of:
- `CV_SUCCESS` `CVodeGetQuad` was successful.
- `CV_MEM_NULL` `cvode_mem` was NULL.
- `CV_NO_QUAD` Quadrature integration was not initialized.
- `CV_BAD_DKY` `yQ` is `NULL`.

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

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

`CVodeGetQuadDky`

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

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

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

Return value The return value `flag` of `CVodeGetQuadDky` is one of:
- `CV_SUCCESS` `CVodeGetQuadDky` succeeded.
- `CV_MEM_NULL` The pointer to `cvode_mem` was NULL.

CV_NO_QUAD Quadrature integration was not initialized.

CV_BAD_DKY The vector dkyQ is NULL.

CV_BAD_K k is not in the range $0, 1, \ldots,$ qlast.

CV_BAD_T The time t is not in the allowed range.

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

4.7.4 Optional inputs for quadrature integration

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

`CVodeSetQuadErrCon`

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

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

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

errconQ (booleantype) specifies whether quadrature variables are included (TRUE) or not (FALSE) in the error control mechanism.

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

CV_SUCCESS The optional value has been successfully set.

CV_MEM_NULL The cvode_mem pointer is NULL.

CV_NO_QUAD Quadrature integration has not been initialized.

Notes By default, errconQ is set to FALSE.

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

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

`CVodeQuadSStolerances`

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

Description The function CVodeQuadSStolerances specifies scalar relative and absolute tolerances.

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

reltolQ (realtype) is the scalar relative error tolerance.

abstolQ (realtype) is the scalar absolute error tolerance.

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

CV_SUCCESS The optional value has been successfully set.

CV_NO_QUAD Quadrature integration was not initialized.

CV_MEM_NULL The cvode_mem pointer is NULL.

CV_ILL_INPUT One of the input tolerances was negative.

`CVodeQuadSVtolerances`

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

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

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

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

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

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

4.7.5 Optional outputs for quadrature integration

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

CVodeGetQuadNumRhsEvals

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

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

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

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

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

CVodeGetQuadNumErrTestFails

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

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

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

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

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

CVodeGetQuadErrWeights

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

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

Arguments
cvode_mem	(void *) pointer to the CVODES memory block.
eQweight	(N_Vector) quadrature error weights at the current time.

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

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

Notes — The user must allocate memory for `eQweight`.

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

`CVodeGetQuadStats`

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

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `nfQevals` (`long int`) number of calls to the user's `fQ` function.
- `nQetfails` (`long int`) number of error test failures due to quadrature variables.

Return value — The return value `flag` (of type `int`) is one of
- `CV_SUCCESS` the optional output values have been successfully set.
- `CV_MEM_NULL` the `cvode_mem` pointer is `NULL`.
- `CV_NO_QUAD` Quadrature integration has not been initialized.

4.7.6 User-supplied function for quadrature integration

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

`CVQuadRhsFn`

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

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

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

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

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

Both `y` and `yQdot` are of type `N_Vector`, but they typically have different internal representations. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with CVODES do not perform any consistency checks with respect to their `N_Vector` arguments (see §7.1 and §7.2).

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

4.8 Preconditioner modules

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

4.8.1 A serial banded preconditioner module

This preconditioner provides a band matrix preconditioner for use with 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 `cvodes_bandpre.h` which declares the needed function prototypes. The following is a summary of the usage of this module. Steps that are unchanged from the skeleton program presented in §4.4 are grayed out.

1. **Set problem dimensions**
2. **Set vector of initial values**
3. **Create CVODES 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 CVODES 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 $-\mathtt{ml} \leq j - i \leq \mathtt{mu}$.

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

`CVBandPrecGetWorkSpace`

Call `flag = CVBandPrecGetWorkSpace(cvode_mem, &lenrwBP, &leniwBP);`

Description The function `CVBandPrecGetWorkSpace` returns the sizes of the CVBANDPRE real and integer workspaces.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `lenrwBP` (`long int`) the number of `realtype` values in the CVBANDPRE workspace.
- `leniwBP` (`long int`) the number of integer values in the CVBANDPRE workspace.

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

- `CVSPILS_SUCCESS` The optional output values have been successfully set.
- `CVSPILS_PMEM_NULL` The CVBANDPRE preconditioner has not been initialized.

Notes In terms of problem size N and $\mathtt{smu} = \min(N - 1, \mathtt{mu+ml})$, the actual size of the real workspace is $(2\,\mathtt{ml} + \mathtt{mu} + \mathtt{smu} + 2)\,N$ `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 CVODES memory block.
- `nfevalsBP` (`long int`) the number of calls to the user right-hand side function.

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

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

`CVSPILS_PMEM_NULL` The CVBANDPRE preconditioner has not been initialized.

Notes The counter `nfevalsBP` is distinct from the counter `nfevalsLS` returned by the corresponding function `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.

4.8.2 A parallel band-block-diagonal preconditioner module

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

However, we have developed one type of preconditioner that treats a rather broad class of PDE-based problems. It has been successfully used for several realistic, large-scale problems [23] and is included in a software module within the CVODES package. This module works with the parallel vector module NVECTOR_PARALLEL and is usable with any of the Krylov iterative linear solvers. 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 CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CVode` returns `CV_LSETUP_FAIL`).

Notes This function must assume that all interprocess communication of data needed to calculate `glocal` has already been done, and that this data is accessible within `user_data`.

The case where g is mathematically identical to f is allowed.

`CVCommFn`

Definition
```
typedef int (*CVCommFn)(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 CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `CVode` returns `CV_LSETUP_FAIL`).

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

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

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

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

1. **Initialize MPI**
2. **Set problem dimensions**
3. **Set vector of initial values**
4. **Create CVODES 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 CVODES 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 CVODES for a subsequent problem, a call to `CVBBDPrecReInit` can be made to change any of the following: the half-bandwidths `mudq` and `mldq` used in the difference-quotient Jacobian approximations, the relative increment `dqrely`, or one of the user-supplied functions `gloc` and `cfn`. If there is a change in any of the linear solver inputs, an additional call to `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 CVODES 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 CVODES memory block.
- `lenrwBBDP` (`long int`) local number of `realtype` values in the CVBBDPRE workspace.
- `leniwBBDP` (`long int`) local number of integer values in the CVBBDPRE workspace.

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

- `CVSPILS_SUCCESS` The optional output value has been successfully set.
- `CVSPILS_MEM_NULL` The `cvode_mem` pointer was `NULL`.
- `CVSPILS_PMEM_NULL` The CVBBDPRE preconditioner has not been initialized.

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

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

- `CVSPILS_SUCCESS` The optional output value has been successfully set.
- `CVSPILS_MEM_NULL` The `cvode_mem` pointer was `NULL`.

CVSPILS_PMEM_NULL The CVBBDPRE preconditioner has not been initialized.

In addition to the ngevalsBBDP gloc evaluations, the costs associated with CVBBDPRE also include nlinsetups LU factorizations, nlinsetups calls to cfn, npsolves banded backsolve calls, and nfevalsLS right-hand side function evaluations, where nlinsetups is an optional CVODES output and npsolves and nfevalsLS are linear solver optional outputs (see §4.5.8).

Chapter 5

Using CVODES for Forward Sensitivity Analysis

This chapter describes the use of CVODES to compute solution sensitivities using forward sensitivity analysis. One of our main guiding principles was to design the CVODES user interface for forward sensitivity analysis as an extension of that for IVP integration. Assuming a user main program and user-defined support routines for IVP integration have already been defined, in order to perform forward sensitivity analysis the user only has to insert a few more calls into the main program and (optionally) define an additional routine which computes the right-hand side of the sensitivity systems (2.11). The only departure from this philosophy is due to the `CVRhsFn` type definition (§4.6.1). Without changing the definition of this type, the only way to pass values of the problem parameters to the ODE right-hand side function is to require the user data structure `f_data` to contain a pointer to the array of real parameters p.

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

We begin with a brief overview, in the form of a skeleton user program. Following that are detailed descriptions of the interface to the various user-callable routines and of the user-supplied routines that were not already described in Chapter 4.

5.1 A skeleton of the user's main program

The following is a skeleton of the user's main program (or calling program) as an application of CVODES. The user program is to have these steps in the order indicated, unless otherwise noted. For the sake of brevity, we defer many of the details to the later sections. As in §4.4, most steps are independent of the NVECTOR implementation used; where this is not the case, usage specifications are given for the four implementations provided with CVODESw8 0 0[(,)-269.657(m)0297743(tn54362(w8 0 0[(,).434975205847(e216(ha)0.443 0

5. **Allocate internal memory**
6. **Specify integration tolerances**
7. **Set optional inputs**
8. **Attach linear solver module**
9. **Set linear solver optional inputs**
10. **Initialize quadrature problem, if not sensitivity-dependent**
11. **Define the sensitivity problem**

    - Number of sensitivities (required)

      Set `Ns` $= N_s$, the number of parameters with respect to which sensitivities are to be computed.

    - Problem parameters (optional)

      If CVODES is to evaluate the right-hand sides of the sensitivity systems, set `p`, an array of `Np` real parameters upon which the IVP depends. Only parameters with respect to which sensitivities are (potentially) desired need to be included. Attach `p` to the user data structure `user_data`. For example, `user_data->p = p;`

      If the user provides a function to evaluate the sensitivity right-hand side, `p` need not be specified.

    - Parameter list (optional)

      If CVODES is to evaluate the right-hand sides of the sensitivity systems, set `plist`, an array of `Ns` integers to specify the parameters `p` with respect to which solution sensitivities are to be computed. If sensitivities with respect to the j-th parameter `p[j]` are desired ($0 \leq$ `j` $<$ `Np`), set $\text{plist}_i = j$, for some $i = 0, \ldots, N_s - 1$.

      If `plist` is not specified, CVODES will compute sensitivities with respect to the first `Ns` parameters; i.e., $\text{plist}_i = i$ $(i = 0, \ldots, N_s - 1)$.

      If the user provides a function to evaluate the sensitivity right-hand side, `plist` need not be specified.

    - Parameter scaling factors (optional)

      If CVODES is to estimate tolerances for the sensitivity solution vectors (based on tolerances for the state solution vector) or if CVODES is to evaluate the right-hand sides of the sensitivity systems using the internal difference-quotient function, the results will be more accurate if order of magnitude information is provided.

      Set `pbar`, an array of `Ns` positive scaling factors. Typically, if $p_i \neq 0$, the value $\bar{p}_i = |p_{\text{plist}_j}|$ can be used.

      If `pbar` is not specified, CVODES will use $\bar{p}_i = 1.0$.

      If the user provides a function to evaluate the sensitivity right-hand side and specifies tolerances for the sensitivity variables, `pbar` need not be specified.

    Note that the names for `p`, `pbar`, `plist`, as well as the field p of `user_data` are arbitrary, but they must agree with the arguments passed to `CVodeSetSensParams` below.

12. **Set sensitivity initial conditions**

    Set the `Ns` vectors `yS0[i]` of initial values for sensitivities (for $i = 0, \ldots,$ `Ns` -1).

    First, create an array of `Ns` vectors by making the call

    [**S**] `yS0 = N_VCloneVectorArray_Serial(Ns, y0);`

    [**O**] `yS0 = N_VCloneVectorArray_OpenMP(Ns, y0);`

    [**T**] `yS0 = N_VCloneVectorArray_Pthreads(Ns, y0);`

[**P**] `yS0 = N_VCloneVectorArray_Parallel(Ns, y0);`

Here the argument `y0` serves only to provide the `N_Vector` type for cloning.

Then, for each $i = 0, \ldots,$`Ns` -1, load initial values for the i-th sensitivity vector into the structure defined by:

[**S**] `NV_DATA_S(yS0[i])`

[**O**] `NV_DATA_OMP(yS0[i])`

[**T**] `NV_DATA_PT(yS0[i])`

[**P**] `NV_DATA_P(yS0[i])`

Alternatively, if the initial values for the sensitivity variables are already available in `realtype` arrays, create an array of `Ns` "empty" vectors by making the call

[**S**] `yS0 = N_VCloneEmptyVectorArray_Serial(Ns, y0);`

[**O**] `yS0 = N_VCloneEmptyVectorArray_OpenMP(Ns, y0);`

[**T**] `yS0 = N_VCloneEmptyVectorArray_Pthreads(Ns, y0);`

[**P**] `yS0 = N_VCloneEmptyVectorArray_Parallel(Ns, y0);`

and then attach the `realtype` array `yS0[i]` containing the initial values of the i-th sensitivity vector using

[**S**] `N_VSetArrayPointer_Serial(yS0_i, yS0[i]);`

[**O**] `N_VSetArrayPointer_OpenMP(yS0_i, yS0[i]);`

[**T**] `N_VSetArrayPointer_Pthreads(yS0_i, yS0[i]);`

[**P**] `N_VSetArrayPointer_Parallel(yS0_i, yS0[i]);`

for $i = 0, \ldots,$`Ns` -1.

13. **Activate sensitivity calculations**

    Call `flag = CVodeSensInit` or `CVodeSensInit1` to activate forward sensitivity computations and allocate internal memory for CVODES related to sensitivity calculations (see §5.2.1).

14. **Set sensitivity tolerances**

    Call `CVodeSensSStolerances` or `CVodeSensSVtolerances` or `CVodeEEtolerances` (see §5.2.2).

15. **Set sensitivity analysis optional inputs**

    Call `CVodeSetSens*` routines to change from their default values any optional inputs that control the behavior of CVODES in computing forward sensitivities. (See §5.2.5.)

16. **Specify rootfinding**

17. **Advance solution in time**

18. **Extract sensitivity solution**

    After each successful return from `CVode`, the solution of the original IVP is available in the `y` argument of `CVode`, while the sensitivity solution can be extracted into `yS` (which can be the same as `yS0`) by calling one of the routines `CVodeGetSens`,`CVodeGetSens1`, `CVodeGetSensDky`, or `CVodeGetSensDky1` (see §5.2.4).

19. **Deallocate memory for solution vector**

20. **Deallocate memory for sensitivity vectors**

    Upon completion of the integration, deallocate memory for the vectors `yS0`:

[S] `N_VDestroyVectorArray_Serial(yS0, Ns);`

[O] `N_VDestroyVectorArray_OpenMP(yS0, Ns);`

[T] `N_VDestroyVectorArray_Pthreads(yS0, Ns);`

[P] `N_VDestroyVectorArray_Parallel(yS0, Ns);`

If `yS` was created from `realtype` arrays `yS_i`, it is the user's responsibility to also free the space for the arrays `yS0_i`.

21. **Free user data structure**
22. **Free solver memory**
23. **Free vector specification memory**

5.2 User-callable routines for forward sensitivity analysis

This section describes the CVODES functions, in addition to those presented in §4.5, that are called by the user to setup and solve a forward sensitivity problem.

5.2.1 Forward sensitivity initialization and deallocation functions

Activation of forward sensitivity computation is done by calling `CVodeSensInit` or `CVodeSensInit1`, depending on whether the sensitivity right-hand side function returns all sensitivities at once or one by one, respectively. The form of the call to each of these routines is as follows:

`CVodeSensInit`

Call `flag = CVodeSensInit(cvode_mem, Ns, ism, fS, yS0);`

Description The routine `CVodeSensInit` activates forward sensitivity computations and allocates internal memory related to sensitivity calculations.

Arguments

- `cvode_mem` (`void *`) pointer to the CVODES memory block returned by `CVodeCreate`.
- `Ns` (`int`) the number of sensitivities to be computed.
- `ism` (`int`) a flag used to select the sensitivity solution method. Its value can be `CV_SIMULTANEOUS` or `CV_STAGGERED`:
  - In the `CV_SIMULTANEOUS` approach, the state and sensitivity variables are corrected at the same time. If `CV_NEWTON` was selected as the nonlinear system solution method, this amounts to performing a modified Newton iteration on the combined nonlinear system;
  - In the `CV_STAGGERED` approach, the correction step for the sensitivity variables takes place at the same time for all sensitivity equations, but only after the correction of the state variables has converged and the state variables have passed the local error test;
- `fS` (`CVSensRhsFn`) is the C function which computes all sensitivity ODE right-hand sides at the same time. For full details see §5.3.
- `yS0` (`N_Vector *`) a pointer to an array of `Ns` vectors containing the initial values of the sensitivities.

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

- `CV_SUCCESS` The call to `CVodeSensInit` was successful.
- `CV_MEM_NULL` The CVODES memory block was not initialized through a previous call to `CVodeCreate`.
- `CV_MEM_FAIL` A memory allocation request has failed.

CV_ILL_INPUT An input argument to CVodeSensInit has an illegal value.

Notes Passing fS=NULL indicates using the default internal difference quotient sensitivity right-hand side routine.

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

It is illegal here to use ism = CV_STAGGERED1. This option requires a different type for fS and can therefore only be used with CVodeSensInit1 (see below).

CVodeSensInit1

Call `flag = CVodeSensInit1(cvode_mem, Ns, ism, fS1, yS0);`

Description The routine CVodeSensInit1 activates forward sensitivity computations and allocates internal memory related to sensitivity calculations.

Arguments
- cvode_mem (void *) pointer to the CVODES memory block returned by CVodeCreate.
- Ns (int) the number of sensitivities to be computed.
- ism (int) a flag used to select the sensitivity solution method. Its value can be CV_SIMULTANEOUS, CV_STAGGERED, or CV_STAGGERED1:
  - In the CV_SIMULTANEOUS approach, the state and sensitivity variables are corrected at the same time. If CV_NEWTON was selected as the nonlinear system solution method, this amounts to performing a modified Newton iteration on the combined nonlinear system;
  - In the CV_STAGGERED approach, the correction step for the sensitivity variables takes place at the same time for all sensitivity equations, but only after the correction of the state variables has converged and the state variables have passed the local error test;
  - In the CV_STAGGERED1 approach, all corrections are done sequentially, first for the state variables and then for the sensitivity variables, one parameter at a time. If the sensitivity variables are not included in the error control, this approach is equivalent to CV_STAGGERED. Note that the CV_STAGGERED1 approach can be used only if the user-provided sensitivity right-hand side function is of type CVSensRhs1Fn (see §5.3).
- fS1 (CVSensRhs1Fn) is the C function which computes the right-hand sides of the sensitivity ODE, one at a time. For full details see §5.3.
- yS0 (N_Vector *) a pointer to an array of Ns vectors containing the initial values of the sensitivities.

Return value The return value flag (of type int) will be one of the following:
- CV_SUCCESS The call to CVodeSensInit1 was successful.
- CV_MEM_NULL The CVODES memory block was not initialized through a previous call to CVodeCreate.
- CV_MEM_FAIL A memory allocation request has failed.
- CV_ILL_INPUT An input argument to CVodeSensInit1 has an illegal value.

Notes Passing fS1=NULL indicates using the default internal difference quotient sensitivity right-hand side routine.

If an error occurred, CVodeSensInit1 also sends an error message to the error handler funciton.

In terms of the problem size N, number of sensitivity vectors N_s, and maximum method order maxord, the size of the real workspace is increased as follows:

- Base value: lenrw = lenrw + (maxord+5)$N_s N$

- With CVodeSensSVtolerances: lenrw = lenrw $+N_s N$

the size of the integer workspace is increased as follows:

- Base value: leniw = leniw + (maxord+5)$N_s N_i$
- With CVodeSensSVtolerances: leniw = leniw $+N_s N_i$

where N_i is the number of integers in one N_Vector.

The routine CVodeSensReInit, useful during the solution of a sequence of problems of same size, reinitializes the sensitivity-related internal memory. The call to it must follow a call to CVodeSensInit or CVodeSensInit1 (and maybe a call to CVodeReInit). The number Ns of sensitivities is assumed to be unchanged since the call to the initialization function. The call to the CVodeSensReInit function has the form:

CVodeSensReInit

Call `flag = CVodeSensReInit(cvode_mem, ism, yS0);`

Description The routine CVodeSensReInit reinitializes forward sensitivity computations.

Arguments
- cvode_mem (void *) pointer to the CVODES memory block returned by CVodeCreate.
- ism (int) a flag used to select the sensitivity solution method. Its value can be CV_SIMULTANEOUS, CV_STAGGERED, or CV_STAGGERED1.
- yS0 (N_Vector *) a pointer to an array of Ns variables of type N_Vector containing the initial values of the sensitivities.

Return value The return value flag (of type int) will be one of the following:
- CV_SUCCESS The call to CVodeReInit was successful.
- CV_MEM_NULL The CVODES memory block was not initialized through a previous call to CVodeCreate.
- CV_NO_SENS Memory space for sensitivity integration was not allocated through a previous call to CVodeSensInit.
- CV_ILL_INPUT An input argument to CVodeSensReInit has an illegal value.
- CV_MEM_FAIL A memory allocation request has failed.

Notes All arguments of CVodeSensReInit are the same as those of the functions CVodeSensInit and CVodeSensInit1.

If an error occurred, CVodeSensReInit also sends a message to the error handler function.

! The value of the input argument ism must be compatible with the type of the sensitivity ODE right-hand side function. Thus if the sensitivity module was initialized using CVodeSensInit, then it is illegal to pass ism = CV_STAGGERED1 to CVodeSensReInit.

To deallocate all forward sensitivity-related memory (allocated in a prior call to CVodeSensInit or CVodeSensInit1), the user must call

CVodeSensFree

Call `CVodeSensFree(cvode_mem);`

Description The function CVodeSensFree frees the memory allocated for forward sensitivity computations by a previous call to CVodeSensInit or CVodeSensInit1.

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

Return value The function CVodeSensFree has no return value.

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

After a call to CVodeSensFree, forward sensitivity computations can be reactivated only by calling CVodeSensInit or CVodeSensInit1 again.

To activate and deactivate forward sensitivity calculations for successive CVODES runs, without having to allocate and deallocate memory, the following function is provided:

`CVodeSensToggleOff`

Call `CVodeSensToggleOff(cvode_mem);`

Description The function `CVodeSensToggleOff` deactivates forward sensitivity calculations. It does *not* deallocate sensitivity-related memory.

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

Return value The return value `flag` of `CVodeSensToggle` is one of:

`CV_SUCCESS` `CVodeSensToggleOff` was successful.

`CV_MEM_NULL` `cvode_mem` was `NULL`.

Notes Since sensitivity-related memory is not deallocated, sensitivities can be reactivated at a later time (using `CVodeSensReInit`).

5.2.2 Forward sensitivity tolerance specification functions

One of the following three functions must be called to specify the integration tolerances for sensitivities. Note that this call must be made after the call to `CVodeSensInit`/`CVodeSensInit1`.

`CVodeSensSStolerances`

Call `flag = CVodeSensSStolerances(cvode_mem, reltolS, abstolS);`

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

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

`reltolS` (`realtype`) is the scalar relative error tolerance.

`abstolS` (`realtype*`) is a pointer to an array of length `Ns` containing the scalar absolute error tolerances, one for each parameter.

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

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

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

`CV_NO_SENS` The sensitivity allocation function (`CVodeSensInit` or `CVodeSensInit1`) has not been called.

`CV_ILL_INPUT` One of the input tolerances was negative.

`CVodeSensSVtolerances`

Call `flag = CVodeSensSVtolerances(cvode_mem, reltolS, abstolS);`

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

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

`reltolS` (`realtype`) is the scalar relative error tolerance.

`abstolS` (`N_Vector*`) is an array of `Ns` variables of type `N_Vector`. The `N_Vector` from `abstolS[is]` specifies the vector tolerances for `is`-th sensitivity.

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

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

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

CV_NO_SENS	The allocation function for sensitivities has not been called.
CV_ILL_INPUT	The relative error tolerance was negative or an absolute tolerance vector had a negative component.

Notes: This choice of tolerances is important when the absolute error tolerance needs to be different for each component of any vector `yS[i]`.

`CVodeSensEEtolerances`

Call: `flag = CVodeSensEEtolerances(cvode_mem);`

Description: When `CVodeSensEEtolerances` is called, CVODES will estimate tolerances for sensitivity variables based on the tolerances supplied for states variables and the scaling factors $\bar{p}$.

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

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

CV_SUCCESS	The call to `CVodeSensEEtolerances` was successful.
CV_MEM_NULL	The CVODES memory block was not initialized through a previous call to `CVodeCreate`.
CV_NO_SENS	The sensitivity allocation function has not been called.

5.2.3 CVODES solver function

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

CV_SRHSFUNC_FAIL	The sensitivity right-hand side function failed in an unrecoverable manner.
CV_FIRST_SRHSFUNC_ERR	The sensitivity right-hand side function failed at the first call.
CV_REPTD_SRHSFUNC_ERR	Convergence tests occurred too many times due to repeated recoverable errors in the sensitivity right-hand side function. This flag will also be returned if the sensitivity right-hand side function had repeated recoverable errors during the estimation of an initial step size.
CV_UNREC_SRHSFUNC_ERR	The sensitivity right-hand function had a recoverable error, but no recovery was possible. This failure mode is rare, as it can occur only if the sensitivity right-hand side function fails recoverably after an error test failed while at order one.

5.2.4 Forward sensitivity extraction functions

If forward sensitivity computations have been initialized by a call to `CVodeSensInit`/`CVodeSensInit1`, or reinitialized by a call to `CVSensReInit`, then CVODES computes both a solution and sensitivities at time `t`. However, `CVode` will still return only the solution y in `yout`. Solution sensitivities can be obtained through one of the following functions:

`CVodeGetSens`

Call: `flag = CVodeGetSens(cvode_mem, &tret, yS);`

Description: The function `CVodeGetSens` returns the sensitivity solution vectors after a successful return from `CVode`.

Arguments:

cvode_mem	(`void *`) pointer to the memory previously allocated by `CVodeInit`.
tret	(`realtype *`) the time reached by the solver (output).
yS	(`N_Vector *`) array of computed forward sensitivity vectors.

Return value: The return value `flag` of `CVodeGetSens` is one of:

CV_SUCCESS	`CVodeGetSens` was successful.

CV_MEM_NULL	cvode_mem was NULL.
CV_NO_SENS	Forward sensitivity analysis was not initialized.
CV_BAD_DKY	yS is NULL.

Notes Note that the argument tret is an output for this function. Its value will be the same as that returned at the last CVode call.

The function CVodeGetSensDky computes the k-th derivatives of the interpolating polynomials for the sensitivity variables at time t. This function is called by CVodeGetSens with $k = 0$, but may also be called directly by the user.

CVodeGetSensDky

Call `flag = CVodeGetSensDky(cvode_mem, t, k, dkyS);`

Description The function CVodeGetSensDky returns derivatives of the sensitivity solution vectors after a successful return from CVode.

Arguments

cvode_mem	(void *) pointer to the memory previously allocated by CVodeInit.
t	(realtype) specifies the time at which sensitivity information is requested. The time t must fall within the interval defined by the last successful step taken by CVODES.
k	(int) order of derivatives.
dkyS	(N_Vector *) array of Ns vectors containing the derivatives on output. The space for dkyS must be allocated by the user.

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

CV_SUCCESS	CVodeGetSensDky succeeded.
CV_MEM_NULL	cvode_mem was NULL.
CV_NO_SENS	Forward sensitivity analysis was not initialized.
CV_BAD_DKY	One of the vectors dkyS is NULL.
CV_BAD_K	k is not in the range $0, 1, ...,$ qlast.
CV_BAD_T	The time t is not in the allowed range.

Forward sensitivity solution vectors can also be extracted separately for each parameter in turn through the functions CVodeGetSens1 and CVodeGetSensDky1, defined as follows:

CVodeGetSens1

Call `flag = CVodeGetSens1(cvode_mem, &tret, is, yS);`

Description The function CVodeGetSens1 returns the is-th sensitivity solution vector after a successful return from CVode.

Arguments

cvode_mem	(void *) pointer to the memory previously allocated by CVodeInit.
tret	(realtype *) the time reached by the solver (output).
is	(int) specifies which sensitivity vector is to be returned ($0 \leq$is$< N_s$).
yS	(N_Vector) the computed forward sensitivity vector.

Return value The return value flag of CVodeGetSens1 is one of:

CV_SUCCESS	CVodeGetSens1 was successful.
CV_MEM_NULL	cvode_mem was NULL.
CV_NO_SENS	Forward sensitivity analysis was not initialized.
CV_BAD_IS	The index is is not in the allowed range.
CV_BAD_DKY	yS is NULL.
CV_BAD_T	The time t is not in the allowed range.

Notes Note that the argument tret is an output for this function. Its value will be the same as that returned at the last CVode call.

`CVodeGetSensDky1`

Call `flag = CVodeGetSensDky1(cvode_mem, t, k, is, dkyS);`

Description The function `CVodeGetSensDky1` returns the `k`-th derivative of the `is`-th sensitivity solution vector after a successful return from `CVode`.

Arguments
- `cvode_mem` (`void *`) pointer to the memory previously allocated by `CVodeInit`.
- `t` (`realtype`) specifies the time at which sensitivity information is requested. The time `t` must fall within the interval defined by the last successful step taken by CVODES.
- `k` (`int`) order of derivative.
- `is` (`int`) specifies the sensitivity derivative vector to be returned ($0 \leq$`is`$< N_s$).
- `dkyS` (`N_Vector`) the vector containing the derivative. The space for `dkyS` must be allocated by the user.

Return value The return value `flag` of `CVodeGetSensDky1` is one of:
- `CV_SUCCESS` `CVodeGetQuadDky1` succeeded.
- `CV_MEM_NULL` The pointer to `cvode_mem` was `NULL`.
- `CV_NO_SENS` Forward sensitivity analysis was not initialized.
- `CV_BAD_DKY` `dkyS` or one of the vectors `dkyS[i]` is `NULL`.
- `CV_BAD_IS` The index `is` is not in the allowed range.
- `CV_BAD_K` `k` is not in the range $0, 1, ...,$ `qlast`.
- `CV_BAD_T` The time `t` is not in the allowed range.

5.2.5 Optional inputs for forward sensitivity analysis

Optional input variables that control the computation of sensitivities can be changed from their default values through calls to `CVodeSetSens*` functions. Table 5.1 lists all forward sensitivity optional input functions in CVODES which are described in detail in the remainder of this section.

`CVodeSetSensParams`

Call `flag = CVodeSetSensParams(cvode_mem, p, pbar, plist);`

Description The function `CVodeSetSensParams` specifies problem parameter information for sensitivity calculations.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `p` (`realtype *`) a pointer to the array of real problem parameters used to evaluate $f(t, y, p)$. If non-`NULL`, `p` must point to a field in the user's data structure `user_data` passed to the right-hand side function. (See §5.1).
- `pbar` (`realtype *`) an array of `Ns` positive scaling factors. If non-`NULL`, `pbar` must have all its components > 0.0. (See §5.1).
- `plist` (`int *`) an array of `Ns` non-negative indices to specify which components `p[i]` to use in estimating the sensitivity equations. If non-`NULL`, `plist` must have all components ≥ 0. (See §5.1).

Table 5.1: Forward sensitivity optional inputs

Optional input	**Routine name**	**Default**
Sensitivity scaling factors	`CVodeSetSensParams`	`NULL`
DQ approximation method	`CVodeSetSensDQMethod`	centered/0.0
Error control strategy	`CVodeSetSensErrCon`	`FALSE`
Maximum no. of nonlinear iterations	`CVodeSetSensMaxNonlinIters`	3

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

`CV_SUCCESS` The optional value has been successfully set.

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

`CV_NO_SENS` Forward sensitivity analysis was not initialized.

`CV_ILL_INPUT` An argument has an illegal value.

Notes This function must be preceded by a call to `CVodeSensInit` or `CVodeSensInit1`.

`CVodeSetSensDQMethod`

Call `flag = CVodeSetSensDQMethod(cvode_mem, DQtype, DQrhomax);`

Description The function `CVodeSetSensDQMethod` specifies the difference quotient strategy in the case in which the right-hand side of the sensitivity equations are to be computed by CVODES.

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

`DQtype` (`int`) specifies the difference quotient type. Its value can be `CV_CENTERED` or `CV_FORWARD`.

`DQrhomax` (`realtype`) positive value of the selection parameter used in deciding switching between a simultaneous or separate approximation of the two terms in the sensitivity right-hand side.

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

`CV_SUCCESS` The optional value has been successfully set.

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

`CV_ILL_INPUT` An argument has an illegal value.

Notes If `DQrhomax` $= 0.0$, then no switching is performed. The approximation is done simultaneously using either centered or forward finite differences, depending on the value of `DQtype`. For values of `DQrhomax` ≥ 1.0, the simultaneous approximation is used whenever the estimated finite difference perturbations for states and parameters are within a factor of `DQrhomax`, and the separate approximation is used otherwise. Note that a value `DQrhomax` < 1.0 will effectively disable switching. See §2.6 for more details.

The default value are `DQtype=CV_CENTERED` and `DQrhomax`$= 0.0$.

`CVodeSetSensErrCon`

Call `flag = CVodeSetSensErrCon(cvode_mem, errconS);`

Description The function `CVodeSetSensErrCon` specifies the error control strategy for sensitivity variables.

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

`errconS` (`booleantype`) specifies whether sensitivity variables are to be included (`TRUE`) or not (`FALSE`) in the error control mechanism.

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

`CV_SUCCESS` The optional value has been successfully set.

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

Notes By default, `errconS` is set to `FALSE`. If `errconS=TRUE` then both state variables and sensitivity variables are included in the error tests. If `errconS=FALSE` then the sensitivity variables are excluded from the error tests. Note that, in any event, all variables are considered in the convergence tests.

`CVodeSetSensMaxNonlinIters`

Call `flag = CVodeSetSensMaxNonlinIters(cvode_mem, maxcorS);`

Description The function `CVodeSetSensMaxNonlinIters` specifies the maximum number of nonlinear solver iterations for sensitivity variables per step.

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

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

Notes The default value is 3.

5.2.6 Optional outputs for forward sensitivity analysis

Optional output functions that return statistics and solver performance information related to forward sensitivity computations are listed in Table 5.2 and described in detail in the remainder of this section.

`CVodeGetSensNumRhsEvals`

Call `flag = CVodeGetSensNumRhsEvals(cvode_mem, &nfSevals);`

Description The function `CVodeGetSensNumRhsEvals` returns the number of calls to the sensitivity right-hand side function.

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

Return value The return value `flag` (of type `int`) is one of:
`CV_SUCCESS` The optional output value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes In order to accommodate any of the three possible sensitivity solution methods, the default internal finite difference quotient functions evaluate the sensitivity right-hand sides one at a time. Therefore, `nfSevals` will always be a multiple of the number of sensitivity parameters (the same as the case in which the user supplies a routine of type `CVSensRhs1Fn`).

Table 5.2: Forward sensitivity optional outputs

Optional output	**Routine name**
No. of calls to sensitivity r.h.s. function	`CVodeGetSensNumRhsEvals`
No. of calls to r.h.s. function for sensitivity	`CVodeGetNumRhsEvalsSens`
No. of sensitivity local error test failures	`CVodeGetSensNumErrTestFails`
No. of calls to lin. solv. setup routine for sens.	`CVodeGetSensNumLinSolvSetups`
Error weight vector for sensitivity variables	`CVodeGetSensErrWeights`
No. of sens. nonlinear solver iterations	`CVodeGetSensNumNonlinSolvIters`
No. of sens. convergence failures	`CVodeGetSensNumNonlinSolvConvFails`
No. of staggered nonlinear solver iterations	`CVodeGetStgrSensNumNonlinSolvIters`
No. of staggered convergence failures	`CVodeGetStgrSensNumNonlinSolvConvFails`

`CVodeGetNumRhsEvalsSens`

Call `flag = CVodeGetNumRhsEvalsSens(cvode_mem, &nfevalsS);`

Description The function `CVodeGetNumRhsEvalsSEns` returns the number of calls to the user's right-hand side function due to the internal finite difference approximation of the sensitivity right-hand sides.

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

`nfevalsS` (`long int`) number of calls to the user's ODE right-hand side function for the evaluation of sensitivity right-hand sides.

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

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

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

`CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes This counter is incremented only if the internal finite difference approximation routines are used for the evaluation of the sensitivity right-hand sides.

`CVodeGetSensNumErrTestFails`

Call `flag = CVodeGetSensNumErrTestFails(cvode_mem, &nSetfails);`

Description The function `CVodeGetSensNumErrTestFails` returns the number of local error test failures for the sensitivity variables that have occurred.

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

`nSetfails` (`long int`) number of error test failures.

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

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

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

`CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes This counter is incremented only if the sensitivity variables have been included in the error test (see `CVodeSetSensErrCon` in §5.2.5). Even in that case, this counter is not incremented if the `ism=CV_SIMULTANEOUS` sensitivity solution method has been used.

`CVodeGetSensNumLinSolvSetups`

Call `flag = CVodeGetSensNumLinSolvSetups(cvode_mem, &nlinsetupsS);`

Description The function `CVodeGetSensNumLinSolvSetups` returns the number of calls to the linear solver setup function due to forward sensitivity calculations.

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

`nlinsetupsS` (`long int`) number of calls to the linear solver setup function.

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

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

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

`CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes This counter is incremented only if Newton iteration has been used and if either the `ism = CV_STAGGERED` or the `ism = CV_STAGGERED1` sensitivity solution method has been specified (see §5.2.1).

`CVodeGetSensStats`

Call `flag = CVodeGetSensStats(cvode_mem, &nfSevals, &nfevalsS, &nSetfails, &nSetfails, &nlinsetupsS);`

Description The function `CVodeGetSensStats` returns all of the above sensitivity-related solver statistics as a group.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `nfSevals` (`long int`) number of calls to the sensitivity right-hand side function.
- `nfevalsS` (`long int`) number of calls to the ODE right-hand side function for sensitivity evaluations.
- `nSetfails` (`long int`) number of error test failures.
- `nlinsetupsS` (`long int`) number of calls to the linear solver setup function.

Return value The return value `flag` (of type `int`) is one of:
- `CV_SUCCESS` The optional output values have been successfully set.
- `CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CV_NO_SENS` Forward sensitivity analysis was not initialized.

`CVodeGetSensErrWeights`

Call `flag = CVodeGetSensErrWeights(cvode_mem, eSweight);`

Description The function `CVodeGetSensErrWeights` returns the sensitivity error weight vectors at the current time. These are the reciprocals of the W_i of (2.7) for the sensitivity variables.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `eSweight` (`N_Vector *`) pointer to the array of error weight vectors.

Return value The return value `flag` (of type `int`) is one of:
- `CV_SUCCESS` The optional output value has been successfully set.
- `CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes The user must allocate memory for `eweightS`.

`CVodeGetSensNumNonlinSolvIters`

Call `flag = CVodeGetSensNumNonlinSolvIters(cvode_mem, &nSniters);`

Description The function `CVodeGetSensNumNonlinSolvIters` returns the number of nonlinear iterations performed for sensitivity calculations.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `nSniters` (`long int`) number of nonlinear iterations performed.

Return value The return value `flag` (of type `int`) is one of:
- `CV_SUCCESS` The optional output value has been successfully set.
- `CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes This counter is incremented only if `ism` was `CV_STAGGERED` or `CV_STAGGERED1` (see §5.2.1).

In the `CV_STAGGERED1` case, the value of `nSniters` is the sum of the number of nonlinear iterations performed for each sensitivity equation. These individual counters can be obtained through a call to `CVodeGetStgrSensNumNonlinSolvIters` (see below).

`CVodeGetSensNumNonlinSolvConvFails`

Call `flag = CVodeGetSensNumNonlinSolvConvFails(cvode_mem, &nSncfails);`

Description The function `CVodeGetSensNumNonlinSolvConvFails` returns the number of nonlinear convergence failures that have occurred for sensitivity calculations.

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

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

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

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

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

`CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes This counter is incremented only if `ism` was `CV_STAGGERED` or `CV_STAGGERED1` (see §5.2.1).

In the `CV_STAGGERED1` case, the value of `nSncfails` is the sum of the number of nonlinear convergence failures that occurred for each sensitivity equation. These individual counters can be obtained through a call to `CVodeGetStgrSensNumNonlinConvFails` (see below).

`CVodeGetSensNonlinSolvStats`

Call `flag = CVodeGetSensNonlinSolvStats(cvode_mem, &nSniters, &nSncfails);`

Description The function `CVodeGetSensNonlinSolvStats` returns the sensitivity-related nonlinear solver statistics as a group.

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

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

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

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

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

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

`CV_NO_SENS` Forward sensitivity analysis was not initialized.

`CVodeGetStgrSensNumNonlinSolvIters`

Call `flag = CVodeGetStgrSensNumNonlinSolvIters(cvode_mem, nSTGR1niters);`

Description The function `CVodeGetStgrSensNumNonlinSolvIters` returns the number of nonlinear (functional or Newton) iterations performed for each sensitivity equation separately, in the `CV_STAGGERED1` case.

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

`nSTGR1niters` (`long int *`) an array (of dimension `Ns`) which will be set with the number of nonlinear iterations performed for each sensitivity system individually.

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

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

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

`CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes The user must allocate space for `nSTGR1niters`.

`CVodeGetStgrSensNumNonlinSolvConvFails`

Call `flag = CVodeGetStgrSensNumNonlinSolvConvFails(cvode_mem, nSTGR1ncfails);`

Description The function `CVodeGetStgrSensNumNonlinSolvConvFails` returns the number of nonlinear convergence failures that have occurred for each sensitivity equation separately, in the `CV_STAGGERED1` case.

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

`nSTGR1ncfails` (`long int *`) an array (of dimension `Ns`) which will be set with the number of nonlinear convergence failures for each sensitivity system individually.

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

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

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

`CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes The user must allocate space for `nSTGR1ncfails`.

5.3 User-supplied routines for forward sensitivity analysis

In addition to the required and optional user-supplied routines described in §4.6, when using CVODES for forward sensitivity analysis, the user has the option of providing a routine that calculates the right-hand side of the sensitivity equations (2.11).

By default, CVODES uses difference quotient approximation routines for the right-hand sides of the sensitivity equations. However, Return value A CVSensRhsFn should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and CV_SRHSFUNC_FAIL is returned).

Notes A sensitivity right-hand side function of type CVSensRhsFn is not compatible with the CV_STAGGERED1 approach.

Allocation of memory for ySdot is handled within CVODES.

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

5.3.2 Sensitivity equations right-hand side (one at a time)

Alternatively, the user may provide the sensitivity right-hand sides, one sensitivity parameter at a time, through a function of type CVSensRhs1Fn. Note that a sensitivity right-hand side function of type CVSensRhs1Fn is compatible with any valid value of the argument ism to CVodeSensInit and CVodeSensInit1, and is *required* if ism = CV_STAGGERED1 in the call to CVodeSensInit1. The type CVSensRhs1Fn is defined by

CVSensRhs1Fn

Definition

```
typedef int (*CVSensRhs1Fn)(int Ns, realtype t,
                            N_Vector y, N_Vector ydot,
                            int iS, N_Vector yS, N_Vector ySdot,
                            void *user_data,
                            N_Vector tmp1, N_Vector tmp2);
```

Purpose This function computes the sensitivity right-hand side for one sensitivity equation at a time. It must compute the vector $(\partial f/\partial y)s_i(t) + (\partial f/\partial p_i)$ for i = iS and store it in ySdot.

Arguments
- t is the current value of the independent variable.
- y is the current value of the state vector, $y(t)$.
- ydot is the current value of the right-hand side of the state equations.
- iS is the index of the parameter for which the sensitivity right-hand side must be computed ($0 \leq$ iS $<$ Ns).
- yS contains the current value of the iS-th sensitivity vector.
- ySdot is the output of CVSensRhs1Fn. On exit it must contain the iS-th sensitivity right-hand side vector.
- user_data is a pointer to user data, the same as the user_data parameter passed to CVodeSetUserData.
- tmp1
- tmp2 are N_Vectors of length N which can be used as temporary storage.

Return value A CVSensRhs1Fn should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and CV_SRHSFUNC_FAIL is returned).

Notes Allocation of memory for ySdot is handled within CVODES.

There are two situations in which recovery is not possible even if CVSensRhs1Fn function returns a recoverable error flag. One is when this occurs at the very first call to the CVSensRhs1Fn (in which case CVODES returns CV_FIRST_SRHSFUNC_ERR). The other is when a recoverable error is reported by CVSensRhs1Fn after an error test failure, while the linear multistep method order equal to 1 (in which case CVODES returns CV_UNREC_SRHSFUNC_ERR).

5.4 Integration of quadrature equations depending on forward sensitivities

CVODES provides support for integration of quadrature equations that depends not only on the state variables but also on forward sensitivities.

The following is an overview of the sequence of calls in a user's main program in this situation. Steps that are unchanged from the skeleton program presented in §5.1 are grayed out.

1. **[P] Initialize MPI**
2. **Set problem dimensions**
3. **Set vectors of initial values**
4. **Create CVODES object**
5. **Allocate internal memory**
6. **Set optional inputs**
7. **Attach linear solver module**
8. **Set linear solver optional inputs**
9. **Initialize sensitivity-independent quadrature problem**
10. **Define the sensitivity problem**
11. **Set sensitivity initial conditions**
12. **Activate sensitivity calculations**
13. **Set sensitivity analysis optional inputs**
14. **Set vector of initial values for quadrature variables**

    Typically, the quadrature variables should be initialized to 0.

15. **Initialize sensitivity-dependent quadrature integration**

    Call CVodeQuadSensInit to specify the quadrature equation right-hand side function and to allocate internal memory related to quadrature integration. See §5.4.1 for details.

16. **Set optional inputs for sensitivity-dependent quadrature integration**

    Call CVodeSetQuadSensErrCon to indicate whether or not quadrature variables should be used in the step size control mechanism. If so, one of the CVodeQuadSens*tolerances functions must be called to specify the integration tolerances for quadrature variables. See §5.4.4 for details.

17. **Advance solution in time**
18. **Extract sensitivity-dependent quadrature variables**

    Call CVodeGetQuadSens, CVodeGetQuadSens1, CVodeGetQuadSensDky or CVodeGetQuadSensDky1 to obtain the values of the quadrature variables or their derivatives at the current time. See §5.4.3 for details.

19. **Get optional outputs**
20. **Extract sensitivity solution**
21. **Get sensitivity-dependent quadrature optional outputs**

    Call `CVodeGetQuadSens*` functions to obtain desired optional output related to the integration of sensitivity-dependent quadratures. See §5.4.5 for details.
22. **Deallocate memory for solutions vector**
23. **Deallocate memory for sensitivity vectors**
24. **Deallocate memory for sensitivity-dependent quadrature variables**
25. **Free solver memory**
26. **[P] Finalize MPI**

Note: `CVodeQuadSensInit` (step 15

- If `CVodeQuadSensSVtolerances` is called: `lenrw` = `lenrw` $+N_q N_s$

and the size of the integer workspace is increased as follows:

- Base value: `leniw` = `leniw` + (`maxord+5`)N_q
- If `CVodeQuadSensSVtolerances` is called: `leniw` = `leniw` $+N_q N_s$

The function `CVodeQuadSensReInit`, useful during the solution of a sequence of problems of same size, reinitializes quadrature-related internal memory and must follow a call to `CVodeQuadSensInit`. The number `Nq` of quadratures as well as the number `Ns` of sensitivities are assumed to be unchanged from the prior call to `CVodeQuadSensInit`. The call to the `CVodeQuadSensReInit` function has the form:

`CVodeQuadSensReInit`

Call `flag = CVodeQuadSensReInit(cvode_mem, yQS0);`

Description The function `CVodeQuadSensReInit` provides required problem specifications and reinitializes the sensitivity-dependent quadrature integration.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `yQS0` (`N_Vector *`) contains the initial values of sensitivity-dependent quadratures.

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

- `CV_SUCCESS` The call to `CVodeQuadSensReInit` was successful.
- `CVODE_MEM_NULL` The CVODES memory was not initialized by a prior call to `CVodeCreate`.
- `CV_NO_SENS` Memory space for the sensitivity calculation was not allocated by a prior call to `CVodeSensInit` or `CVodeSensInit1`.
- `CV_NO_QUADSENS` Memory space for the sensitivity quadratures integration was not allocated by a prior call to `CVodeQuadSensInit`.
- `CV_ILL_INPUT` The parameter `yQS0` is `NULL`.

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

`CVodeQuadSensFree`

Call `CVodeQuadSensFree(cvode_mem);`

Description The function `CVodeQuadSensFree` frees the memory allocated for sensitivity quadrature integration.

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

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

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

5.4.2 CVODES solver function

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

- `CV_QSRHSFUNC_ERR` The sensitivity quadrature right-hand side function failed in an unrecoverable manner.
- `CV_FIRST_QSRHSFUNC_ERR` The sensitivity quadrature right-hand side function failed at the first call.

CV_REPTD_QSRHSFUNC_ERR Convergence test failures occurred too many times due to repeated recoverable errors in the quadrature right-hand side function. This flag will also be returned if the quadrature right-hand side function had repeated recoverable errors during the estimation of an initial step size (assuming the sensitivity quadrature variables are included in the error tests).

5.4.3 Sensitivity-dependent quadrature extraction functions

If sensitivity-dependent quadratures have been initialized by a call to `CVodeQuadSensInit`, or reinitialized by a call to `CVodeQuadSensReInit`, then CVODES computes a solution, sensitivity vectors, and quadratures depending on sensitivities at time `t`. However, `CVode` will still return only the solution y. Sensitivity-dependent quadratures can be obtained using one of the following functions:

`CVodeGetQuadSens`

Call `flag = CVodeGetQuadSens(cvode_mem, &tret, yQS);`

Description The function `CVodeGetQuadSens` returns the quadrature sensitivities solution vectors after a successful return from `CVode`.

Arguments `cvode_mem` (`void *`) pointer to the memory previously allocated by `CVodeInit`.

`tret` (`realtype`

CV_BAD_T	The time `t` is not in the allowed range.

Quadrature sensitivity solution vectors can also be extracted separately for each parameter in turn through the functions `CVodeGetQuadSens1` and `CVodeGetQuadSensDky1`, defined as follows:

CVodeGetQuadSens1

Call `flag = CVodeGetQuadSens1(cvode_mem, &tret, is, yQS);`

Description The function `CVodeGetQuadSens1` returns the `is`-th sensitivity of quadratures after a successful return from `CVode`.

Arguments
- `cvode_mem` (`void *`) pointer to the memory previously allocated by `CVodeInit`.
- `tret` (`realtype`) the time reached by the solver (output).
- `is` (`int`) specifies which sensitivity vector is to be returned ($0 \leq$ `is` $< N_s$).
- `yQS` (`N_Vector`) the computed sensitivity-dependent quadrature vector.

Return value The return value `flag` of `CVodeGetQuadSens1` is one of:

CV_SUCCESS	`CVodeGetQuadSens1` was successful.
CVODE_MEM_NULL	`cvode_mem` was `NULL`.
CV_NO_SENS	Forward sensitivity analysis was not initialized.
CV_NO_QUADSENS	Quadratures depending on the sensitivities were not activated.
CV_BAD_IS	The index `is` is not in the allowed range.
CV_BAD_DKY	`yQS` is `NULL`.

CVodeGetQuadSensDky1

Call `flag = CVodeGetQuadSensDky1(cvode_mem, t, k, is, dkyQS);`

Description The function `CVodeGetQuadSensDky1` returns the `k`-th derivative of the `is`-th sensitivity solution vector after a successful return from `CVode`.

Arguments
- `cvode_mem` (`void *`) pointer to the memory previously allocated by `CVodeInit`.
- `t` (`realtype`) specifies the time at which sensitivity information is requested. The time `t` must fall within the interval defined by the last successful step taken by CVODES.
- `k` (`int`) order of derivative.
- `is` (`int`) specifies the sensitivity derivative vector to be returned ($0 \leq$`is`$< N_s$).
- `dkyQS` (`N_Vector`) the vector containing the derivative on output. The space for `dkyQS` must be allocated by the user.

Return value The return value `flag` of `CVodeGetQuadSensDky1` is one of:

CV_SUCCESS	`CVodeGetQuadDky1` succeeded.
CVODE_MEM_NULL	`cvode_mem` was `NULL`.
CV_NO_SENS	Forward sensitivity analysis was not initialized.
CV_NO_QUADSENS	Quadratures depending on the sensitivities were not activated.
CV_BAD_DKY	`dkyQS` is `NULL`.
CV_BAD_IS	The index `is` is not in the allowed range.
CV_BAD_K	`k` is not in the range $0, 1, ...,$ `qlast`.
CV_BAD_T	The time `t` is not in the allowed range.

5.4.4 Optional inputs for sensitivity-dependent quadrature integration

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

`CVodeSetQuadSensErrCon`

Call `flag = CVodeSetQuadSensErrCon(cvode_mem, errconQS)`

Description The function `CVodeSetQuadSensErrCon` specifies whether or not the quadrature variables are to be used in the step size control mechanism. If they are, the user must call one of the functions `CVodeQuadSensSStolerances`, `CVodeQuadSensSVtolerances`, or `CVodeQuadSensEEtolerances` to specify the integration tolerances for the quadrature variables.

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

`errconQS` (`booleantype`) specifies whether sensitivity quadrature variables are to be included (`TRUE`) or not (`FALSE`) in the error control mechanism.

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

`CV_SUCCESS`	The optional value has been successfully set.
`CVODE_MEM_NULL`	`cvode_mem` is `NULL`.
`CV_NO_SENS`	Sensitivities were not activated.
`CV_NO_QUADSENS`	Quadratures depending on the sensitivities were not activated.

Notes By default, `errconQS` is set to `FALSE`.

It is illegal to call `CVodeSetQuadSensErrCon` before a call to `CVodeQuadSensInit`.

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

`CVodeQuadSensSStolerances`

Call `flag = CVodeQuadSensSVtolerances(cvode_mem, reltolQS, abstolQS);`

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

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

`reltolQS` (`realtype`) is the scalar relative error tolerance.

`abstolQS` (`realtype*`) is a pointer to an array containing the `Ns` scalar absolute error tolerances.

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

`CV_SUCCESS`	The optional value has been successfully set.
`CVODE_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CV_NO_SENS`	Sensitivities were not activated.
`CV_NO_QUADSENS`	Quadratures depending on the sensitivities were not activated.
`CV_ILL_INPUT`	One of the input tolerances was negative.

`CVodeQuadSensSVtolerances`

Call `flag = CVodeQuadSensSVtolerances(cvode_mem, reltolQS, abstolQS);`

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

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

`reltolQS` (`realtype`) is the scalar relative error tolerance.

`abstolQS` (`N_Vector*`) is an array of `Ns` variables of type `N_Vector`. The `N_Vector` `abstolS[is]` specifies the vector tolerances for `is`-th quadrature sensitivity.

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

`CV_SUCCESS`	The optional value has been successfully set.
`CV_NO_QUAD`	Quadrature integration was not initialized.

`CVODE_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CV_NO_SENS`	Sensitivities were not activated.
`CV_NO_QUADSENS`	Quadratures depending on the sensitivities were not activated.
`CV_ILL_INPUT`	One of the input tolerances was negative.

`CVodeQuadSensEEtolerances`

Call `flag = CVodeQuadSensEEtolerances(cvode_mem);`

Description A call to the function `CVodeQuadSensEEtolerances` specifies that the tolerances for the sensitivity-dependent quadratures should be estimated from those provided for the pure quadrature variables.

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

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

`CV_SUCCESS`	The optional value has been successfully set.
`CVODE_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CV_NO_SENS`	Sensitivities were not activated.
`CV_NO_QUADSENS`	Quadratures depending on the sensitivities were not activated.

Notes When `CVodeQuadSensEEtolerances` is used, before calling `CVode`, integration of pure quadratures must be initialized (see 4.7.1) and tolerances for pure quadratures must be also specified (see 4.7.4).

5.4.5 Optional outputs for sensitivity-dependent quadrature integration

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

`CVodeGetQuadSensNumRhsEvals`

Call `flag = CVodeGetQuadSensNumRhsEvals(cvode_mem, &nrhsQSevals);`

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

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

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

`CV_SUCCESS`	The optional output value has been successfully set.
`CVODE_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CV_NO_QUADSENS`	Sensitivity-dependent quadrature integration has not been initialized.

`CVodeGetQuadSensNumErrTestFails`

Call `flag = CVodeGetQuadSensNumErrTestFails(cvode_mem, &nQSetfails);`

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

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

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

`CV_SUCCESS`	The optional output value has been successfully set.
`CVODE_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CV_NO_QUADSENS`	Sensitivity-dependent quadrature integration has not been initialized.

`CVodeGetQuadSensErrWeights`

Call `flag = CVodeGetQuadSensErrWeights(cvode_mem, eQSweight);`

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `eQSweight` (`N_Vector *`) array of quadrature error weight vectors at the current time.

Return value The return value `flag` (of type `int`) is one of:
- `CV_SUCCESS` The optional output value has been successfully set.
- `CVODE_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CV_NO_QUADSENS` Sensitivity-dependent quadrature integration has not been initialized.

Notes The user must allocate memory for `eQSweight`.

If quadratures were not included in the error control mechanism (through a call to `CVodeSetQuadSensErrCon` with `errconQS = TRUE`), then this function does not set the `eQSweight` array.

`CVodeGetQuadSensStats`

Call `flag = CVodeGetQuadSensStats(cvode_mem, &nrhsQSevals, &nQSetfails);`

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `nrhsQSevals` (`long int`) number of calls to the user's `rhsQS` function.
- `nQSetfails` (`long int`) number of error test failures due to quadrature variables.

Return value The return value `flag` (of type `int`) is one of
- `CV_SUCCESS` the optional output values have been successfully set.
- `CVODE_MEM_NULL` the `cvode_mem` pointer is `NULL`.
- `CV_NO_QUADSENS` Sensitivity-dependent quadrature integration has not been initialized.

5.4.6 User-supplied function for sensitivity-dependent quadrature integration

For the integration of sensitivity-dependent quadrature equations, the user must provide a function that defines the right-hand side of those quadrature equations. For the sensitivities of quadratures (2.9) with integrand q, the appropriate right-hand side functions are given by: $\bar{q}_i = q_y s_i + q_{p_i}$. This user function must be of type `CVQuadSensRhsFn` defined as follows:

`CVQuadSensRhsFn`

Definition
```
typedef int (*CVQuadSensRhsFn)(int Ns, realtype t, N_Vector y,
                               N_Vector yS, N_Vector yQdot,
                               N_Vector *rhsvalQS, void *user_data,
                               N_Vector tmp1, N_Vector tmp2)
```

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

Arguments
- `Ns` is the number of sensitivity vectors.
- `t` is the current value of the independent variable.
- `y` is the current value of the dependent variable vector, $y(t)$.
- `yS` is an array of `Ns` variables of type `N_Vector` containing the dependent sensitivity vectors s_i.

yQdot	is the current value of the quadrature right-hand side, q.
rhsvalQS	array of Ns vectors to contain the right-hand sides.
user_data	is the user_data pointer passed to CVodeSetUserData.
tmp1	
tmp2	are N_Vectors which can be used as temporary storage.

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

Notes Allocation of memory for rhsvalQS is automatically handled within CVODES.

Here y is of type N_Vector and yS is a pointer to an array containing Ns vectors of type N_Vector. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with CVODES do not perform any consistency checks with respect to their N_Vector arguments (see §7.1 and §7.2).

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

5.5 Note on using partial error control

For some problems, when sensitivities are excluded from the error control test, the behavior of CVODES may appear at first glance to be erroneous. One would expect that, in such cases, the sensitivity variables would not influence in any way the step size selection. A comparison of the solver diagnostics reported for cvsdenx and the second run of the cvsfwddenx example in [32] indicates that this may not always be the case.

The short explanation of this behavior is that the step size selection implemented by the error control mechanism in CVODES is based on the magnitude of the correction calculated by the nonlinear solver. As mentioned in §5.2.1, even with partial error control selected (in the call to CVodeSetSensErrCon), the sensitivity variables are included in the convergence tests of the nonlinear solver.

When using the simultaneous corrector method (§2.6), the nonlinear system that is solved at each step involves both the state and sensitivity equations. In this case, it is easy to see how the sensitivity variables may affect the convergence rate of the nonlinear solver and therefore the step size selection. The case of the staggered corrector approach is more subtle. After all, in this case (ism = CV_STAGGERED or CV_STAGGERED1 in the call to CVodeSensInit/CVodeSensInit1), the sensitivity variables at a given step are computed only once the solver for the nonlinear state equations has converged. However, if the nonlinear system corresponding to the sensitivity equations has convergence problems, CVODES will attempt to improve the initial guess by reducing the step size in order to provide a better prediction of the sensitivity variables. Moreover, even if there are no convergence failures in the solution of the sensitivity system, CVODES may trigger a call to the linear solver's setup routine which typically involves reevaluation of Jacobian information (Jacobian approximation in the case of CVDENSE and CVBAND, or preconditioner data in the case of the Krylov solvers). The new Jacobian information will be used by subsequent calls to the nonlinear solver for the state equations and, in this way, potentially affect the step size selection.

When using the simultaneous corrector method it is not possible to decide whether nonlinear solver convergence failures or calls to the linear solver setup routine have been triggered by convergence problems due to the state or the sensitivity equations. When using one of the staggered corrector methods however, these situations can be identified by carefully monitoring the diagnostic information provided through optional outputs. If there are no convergence failures in the sensitivity nonlinear solver, and none of the calls to the linear solver setup routine were made by the sensitivity nonlinear solver, then the step size selection is not affected by the sensitivity variables.

Finally, the user must be warned that the effect of appending sensitivity equations to a given system of ODEs on the step size selection (through the mechanisms described above) is problem-dependent and can therefore lead to either an increase or decrease of the total number of steps that CVODES takes to complete the simulation. At first glance, one would expect that the impact of the sensitivity variables, if any, would be in the direction of increasing the step size and therefore reducing the total number of steps. The argument for this is that the presence of the sensitivity variables in the convergence test of the nonlinear solver can only lead to additional iterations (and therefore a smaller final iteration error), or to additional calls to the linear solver setup routine (and therefore more up-to-date Jacobian information), both of which will lead to larger steps being taken by CVODES. However, this is true only locally. Overall, a larger integration step taken at a given time may lead to step size reductions at later times, due to either nonlinear solver convergence failures or error test failures.

Chapter 6

Using CVODES for Adjoint Sensitivity Analysis

This chapter describes the use of CVODES to compute sensitivities of derived functions using adjoint sensitivity analysis. As mentioned before, the adjoint sensitivity module of CVODES provides the infrastructure for integrating backward in time any system of ODEs that depends on the solution of the original IVP, by providing various interfaces to the main CVODES integrator, as well as several supporting user-callable functions. For this reason, in the following sections we refer to the *backward problem* and not to the *adjoint problem* when discussing details relevant to the ODEs that are integrated backward in time. The backward problem can be the adjoint problem (2.19) or (2.22), and can be augmented with some quadrature differential equations.

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

We begin with a brief overview, in the form of a skeleton user program. Following that are detailed descriptions of the interface to the various user-callable functions and of the user-supplied functions that were not already described in Chapter 4.

6.1 A skeleton of the user's main program

The following is a skeleton of the user's main program as an application of CVODES. The user program is to have these steps in the order indicated, unless otherwise noted. For the sake of brevity, we defer many of the details to the later sections. As in §4.4, most steps are independent of the NVECTOR implementation used; where this is not the case, usage specifications are given for the two implementations provided with CVODES: steps marked [**P**] correspond to NVECTOR_PARALLEL, steps marked [**O**] correspond to NVECTOR_OPENMP, steps marked [**T**] correspond to NVECTOR_PTHREADS, while steps marked [**S**] correspond to NVECTOR_SERIAL. Steps that are unchanged from the skeleton programs presented in §4.4, §5.1, and §5.4, are grayed out.

1. **Include necessary header files**

   The `cvodes.h` header file also defines additional types, constants, and function prototypes for the adjoint sensitivity module user-callable functions. In addition, the main program should include an NVECTOR implementation header file (`nvector_serial.h`, `nvector_openmp.h`, `nvector_pthreads.h`, or `nvector_parallel.h` for the implementations provided with CVODES) and, if Newton iteration was selected, the main header file of the desired linear solver module.

2. **[P] Initialize MPI**

Forward problem

3. **Set problem dimensions for the forward problem**

4. **Set initial conditions for the forward problem**

5. **Create CVODES object for the forward problem**

6. **Allocate internal memory for the forward problem**

7. **Specify integration tolerances for forward problem**

8. **Set optional inputs for the forward problem**

9. **Attach linear solver module for the forward problem**

10. **Set linear solver optional inputs for the forward problem**

11. **Initialize quadrature problem or problems for forward problems, using CVodeQuadInit and/or CVodeQuadSensInit.**

12. **Initialize forward sensitivity problem**

13. **Allocate space for the adjoint computation**

    Call `CVodeAdjInit()` to allocate memory for the combined forward-backward problem (see §6.2.1 for details). This call requires `Nd`, the number of steps between two consecutive checkpoints. `CVodeAdjInit` also specifies the type of interpolation used (see §2.7.1).

14. **Integrate forward problem**

    Call `CVodeF`, a wrapper for the CVODES main integration function `CVode`, either in `CV_NORMAL` mode to the time `tout` or in `CV_ONE_STEP` mode inside a loop (if intermediate solutions of the forward problem are desired (see §6.2.2)). The final value of `tret` is then the maximum allowable value for the endpoint T of the backward problem.

Backward problem(s)

15. **Set problem dimensions for the backward problem**

    [**S**], [**O**], [**T**] set `NB`, the number of variables in the backward problem
    [**P**] set `NB` and `NBlocal`
    [**O**], [**T**] set `num_threads`

16. **Set initial values for the backward problem**

    Set the endpoint time `tB0` $= T$ and the corresponding vector `yB0` at which the backward problem starts.

17. **Create the backward problem**

    Call `CVodeCreateB`, a wrapper for `CVodeCreate`, to create the CVODES memory block for the new backward problem. Unlike `CVodeCreate`, the function `CVodeCreateB` does not return a pointer to the newly created memory block (see §6.2.3). Instead, this pointer is attached to the internal adjoint memory block (created by `CVodeAdjInit`) and returns an identifier called `which` that the user must later specify in any actions on the newly created backward problem.

18. **Allocate memory for the backward problem**

    Call `CVodeInitB` (or `CVodeInitBS`, when the backward problem depends on the forward sensitivities). The two functions are actually wrappers for `CVodeInit` and allocate internal memory, specify problem data, and initialize CVODES at `tB0` for the backward problem (see §6.2.3).

19. **Specify integration tolerances for backward problem**

Call `CVodeSStolerancesB(...)` or `CVodeSVtolerancesB(...)` to specify a scalar relative tolerance and scalar absolute tolerance or scalar relative tolerance and a vector of absolute tolerances, respectively. The functions are wrappers for `CVodeSStolerances` and `CVodeSVtolerances`, but they require an extra argument `which`, the identifier of the backward problem returned by `CVodeCreateB`. See §6.2.4 for more information.

20. **Set optional inputs for the backward problem**

    Call `CVodeSet*B` functions to change from their default values any optional inputs that control the behavior of CVODES. Unlike their counterparts for the forward problem, these functions take an extra argument `which`, the identifier of the backward problem returned by `CVodeCreateB` (see §6.2.8).

21. **Attach linear solver module for the backward problem**

    Initialize the linear solver module for the backward problem by calling the appropriate wrapper function: `CVDenseB`, `CVBandB`, `CVLapackDenseB`, `CVLapackBandB`, `CVDiagB`, `CVKLUB`, `CVSuperLUMTB`, `CVodeSpgmrB`, `CVodeSpbcgB`, or `CVodeSptfqmrB` (see §6.2.5). Note that it is not required to use the same linear solver module for both the forward and the backward problems; for example, the forward problem could be solved with the CVDENSE linear solver and the backward problem with CVSPGMR.

22. **Initialize quadrature calculation**

    If additional quadrature equations must be evaluated, call `CVodeQuadInitB` or `CVodeQuadInitBS` (if quadrature depends also on the forward sensitivities) as shown in §6.2.10.1. These functions are wrappers around `CVodeQuadInit` and can be used to initialize and allocate memory for quadrature integration. Optionally, call `CVodeSetQuad*B` functions to change from their default values optional inputs that control the integration of quadratures during the backward phase.

23. **Integrate backward problem**

    Call `CVodeB`, a second wrapper around the CVODES main integration function `CVode`, to integrate the backward problem from `tB0` (see §6.2.6). This function can be called either in `CV_NORMAL` or `CV_ONE_STEP` mode. Typically, `CVodeB` will be called in `CV_NORMAL` mode with an end time equal to the initial time t_0 of the forward problem.

24. **Extract quadrature variables**

    If applicable, call `CVodeGetQuadB`, a wrapper around `CVodeGetQuad`, to extract the values of the quadrature variables at the time returned by the last call to `CVodeB`. See §6.2.10.2.

25. **Deallocate memory**

    Upon completion of the backward integration, call all necessary deallocation functions. These include appropriate destructors for the vectors `y` and `yB`, a call to `CVodeFree` to free the CVODES memory block for the forward problem. If additional forward integration(s) are to be done for this problem, a call to `CVodeAdjFree` (see §6.2.1) may be made to free and deallocate memory allocated for the backward problems.

26. **Finalize MPI**

    [**P**] If MPI was initialized by the user main program, call `MPI_Finalize();`.

The above user interface to the adjoint sensitivity module in CVODES was motivated by the desire to keep it as close as possible in look and feel to the one for ODE IVP integration. Note that if steps (15)-(24) are not present, a program with the above structure will have the same functionality as one described in §4.4 for integration of ODEs, albeit with some overhead due to the checkpointing scheme.

If there are multiple backward problems associated with the same forward problem, repeat steps (15)-(24) above for each successive backward problem. In the process, each call to `CVodeCreateB` creates a new value of the identifier `which`.

6.2 User-callable functions for adjoint sensitivity analysis

6.2.1 Adjoint sensitivity allocation and deallocation functions

After the setup phase for the forward problem, but before the call to `CVodeF`, memory for the combined forward-backward problem must be allocated by a call to the function `CVodeAdjInit`. The form of the call to this function is

`CVodeAdjInit`

Call `flag = CVodeAdjInit(cvode_mem, Nd, interpType);`

Description The function `CVodeAdjInit` updates CVODES memory block by allocating the internal memory needed for backward integration. Space is allocated for the `Nd` $= N_d$ interpolation data points, and a linked list of checkpoints is initialized.

Arguments
- `cvode_mem` (`void *`) is the pointer to the CVODES memory block returned by a previous call to `CVodeCreate`.
- `Nd` (`long int`) is the number of integration steps between two consecutive checkpoints.
- `interpType` (`int`) specifies the type of interpolation used and can be `CV_POLYNOMIAL` or `CV_HERMITE`, indicating variable-degree polynomial and cubic Hermite interpolation, respectively (see §2.7.1).

Return value The return value `flag` (of type `int`) is one of:
- `CV_SUCCESS` `CVodeAdjInit` was successful.
- `CV_MEM_FAIL` A memory allocation request has failed.
- `CV_MEM_NULL` `cvode_mem` was NULL.
- `CV_ILL_INPUT` One of the parameters was invalid: `Nd` was not positive or `interpType` is not one of the `CV_POLYNOMIAL` or `CV_HERMITE`.

Notes The user must set `Nd` so that all data needed for interpolation of the forward problem solution between two checkpoints fits in memory. `CVodeAdjInit` attempts to allocate space for (2`Nd`+3) variables of type `N_Vector`.

If an error occurred, `CVodeAdjInit` also sends a message to the error handler function.

`CVodeAdjReInit`

Call `flag = CVodeAdjReInit(cvode_mem);`

Description The function `CVodeAdjReInit` reinitializes the CVODES memory block for ASA, assuming that the number of steps between check points and the type of interpolation remain unchanged.

Arguments
- `cvode_mem` (`void *`) is the pointer to the CVODES memory block returned by a previous call to `CVodeCreate`.

Return value The return value `flag` (of type `int`) is one of:
- `CV_SUCCESS` `CVodeAdjReInit` was successful.
- `CV_MEM_NULL` `cvode_mem` was NULL.
- `CV_NO_ADJ` The function `CVodeAdjInit` was not previously called.

Notes The list of check points (and associated memory) is deleted.

The list of backward problems is kept. However, new backward problems can be added to this list by calling `CVodeCreateB`. If a new list of backward problems is also needed, then free the adjoint memory (by calling `CVodeAdjFree`) and reinitialize ASA with `CVodeAdjInit`.

The CVODES memory for the forward and backward problems can be reinitialized separately by calling `CVodeReInit` and `CVodeReInitB`, respectively.

`CVodeAdjFree`

Call `CVodeAdjFree(cvode_mem);`

Description The function `CVodeAdjFree` frees the memory related to backward integration allocated by a previous call to `CVodeAdjInit`.

Arguments The only argument is the CVODES memory block pointer returned by a previous call to `CVodeCreate`.

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

Notes This function frees all memory allocated by `CVodeAdjInit`. This includes workspace memory, the linked list of checkpoints, memory for the interpolation data, as well as the CVODES

CV_CONV_FAILURE Convergence test failures occurred too many times during one internal time step or occurred with $|h| = h_{min}$.

CV_LSETUP_FAIL The linear solver's setup function failed in an unrecoverable manner.

CV_LSOLVE_FAIL The linear solver's solve function failed in an unrecoverable manner.

CV_NO_ADJ The function CVodeAdjInit has not been previously called.

CV_MEM_FAIL A memory allocation request has failed (in an attempt to allocate space for a new checkpoint).

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

At this time, CVodeF stores checkpoint information in memory only. Future versions will provide for a safeguard option of dumping checkpoint data into a temporary file as needed. The data stored at each checkpoint is basically a snapshot of the CVODES internal memory block and contains enough information to restart the integration from that time and to proceed with the same step size and method order sequence as during the forward integration.

In addition, CVodeF also stores interpolation data between consecutive checkpoints so that, at the end of this first forward integration phase, interpolation information is already available from the last checkpoint forward. In particular, if no checkpoints were necessary, there is no need for the second forward integration phase.

It is illegal to change the integration tolerances between consecutive calls to CVodeF, as this information is not captured in the checkpoint data.

6.2.3 Backward problem initialization functions

The functions CVodeCreateB and CVodeInitB (or CVodeInitBS) must be called in the order listed. They instantiate a CVODES solver object, provide problem and solution specifications, and allocate internal memory for the backward problem.

CVodeCreateB

Call `flag = CVodeCreateB(cvode_mem, lmmB, iterB, &which);`

Description The function CVodeCreateB instantiates a CVODES solver object and specifies the solution method for the backward problem.

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

lmmB (int) specifies the linear multistep method and may be one of two possible values: CV_ADAMS or CV_BDF.

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

which (int) contains the identifier assigned by CVODES for the newly created backward problem. Any call to CVode*B functions requires such an identifier.

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

CV_SUCCESS The call to CVodeCreateB was successful.

CV_MEM_NULL cvode_mem was NULL.

CV_NO_ADJ The function CVodeAdjInit has not been previously called.

CV_MEM_FAIL A memory allocation request has failed.

There are two initialization functions for the backward problem – one for the case when the backward problem does not depend on the forward sensitivities, and one for the case when it does. These two functions are described next.

The function CVodeInitB initializes the backward problem when it does not depend on the forward sensitivities. It is essentially a wrapper for CVodeInit with some particularization for backward integration, as described below.

`CVodeInitB`

Call `flag = CVodeInitB(cvode_mem, which, rhsB, tB0, yB0);`

Description The function `CVodeInitB` provides problem specification, allocates internal memory, and initializes the backward problem.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block returned by `CVodeCreate`.
- `which` (`int`) represents the identifier of the backward problem.
- `rhsB` (`CVRhsFnB`) is the C function which computes fB, the right-hand side of the backward ODE problem. This function has the form `rhsB(t, y, yB, yBdot, user_dataB)` (for full details see §6.3.1).
- `tB0` (`realtype`) specifies the endpoint T where final conditions are provided for the backward problem, normally equal to the endpoint of the forward integration.
- `yB0` (`N_Vector`) is the initial value (at $t =$ `tB0`) of the backward solution.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeInitB` was successful.
- `CV_NO_MALLOC` The function `CVodeInit` has not been previously called.
- `CV_MEM_NULL` `cvode_mem` was `NULL`.
- `CV_NO_ADJ` The function `CVodeAdjInit` has not been previously called.
- `CV_BAD_TB0` The final time `tB0` was outside the interval over which the forward problem was solved.
- `CV_ILL_INPUT` The parameter `which` represented an invalid identifier, or either `yB0` or `rhsB` was `NULL`.

Notes The memory allocated by `CVodeInitB` is deallocated by the function `CVodeAdjFree`.

For the case when backward problem also depends on the forward sensitivities, user must call `CVodeInitBS` instead of `CVodeInitB`. Only the third argument of each function differs between these two functions.

`CVodeInitBS`

Call `flag = CVodeInitBS(cvode_mem, which, rhsBS, tB0, yB0);`

Description The function `CVodeInitBS` provides problem specification, allocates internal memory, and initializes the backward problem.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block returned by `CVodeCreate`.
- `which` (`int`) represents the identifier of the backward problem.
- `rhsBS` (`CVRhsFnBS`) is the C function which computes fB, the right-hand side of the backward ODE problem. This function has the form `rhsBS(t, y, yS, yB, yBdot, user_dataB)` (for full details see §6.3.2).
- `tB0` (`realtype`) specifies the endpoint T where final conditions are provided for the backward problem.
- `yB0` (`N_Vector`) is the initial value (at $t =$ `tB0`) of the backward solution.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeInitB` was successful.
- `CV_NO_MALLOC` The function `CVodeInit` has not been previously called.
- `CV_MEM_NULL` `cvode_mem` was `NULL`.
- `CV_NO_ADJ` The function `CVodeAdjInit` has not been previously called.
- `CV_BAD_TB0` The final time `tB0` was outside the interval over which the forward problem was solved.

CV_ILL_INPUT	The parameter which represented an invalid identifier, either yB0 or rhsBS was NULL, or sensitivities were not active during the forward integration.

Notes: The memory allocated by CVodeInitBS is deallocated by the function CVodeAdjFree.

The function CVodeReInitB reinitializes CVODES for the solution of a series of backward problems, each identified by a value of the parameter which. CVodeReInitB is essentially a wrapper for CVodeReInit, and so all details given for CVodeReInit in §4.5.9 apply. Also note that CVodeReInitB can be called to reinitialize the backward problem even it has been initialized with the sensitivity-dependent version CVodeInitBS. The call to the CVodeReInitB function has the form

CVodeReInitB

Call: `flag = CVodeReInitB(cvode_mem, which, tB0, yB0)`

Description: The function CVodeReInitB reinitializes CVODES the backward problem.

Arguments:

cvode_mem	(void *) pointer to CVODES memory block returned by CVodeCreate.
which	(int) represents the identifier of the backward problem.
tB0	(realtype) specifies the endpoint T where final conditions are provided for the backward problem.
yB0	(N_Vector) is the initial value (at $t =$ tB0) of the backward solution.

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

CV_SUCCESS	The call to CVodeReInitB was successful.
CV_NO_MALLOC	The function CVodeInit has not been previously called.
CV_MEM_NULL	The cvode_mem memory block pointer was NULL.
CV_NO_ADJ	The function CVodeAdjInit has not been previously called.
CV_BAD_TB0	The final time tB0 is outside the interval over which the forward problem was solved.
CV_ILL_INPUT	The parameter which represented an invalid identifier, or yB0 was NULL.

6.2.4 Tolerance specification functions for backward problem

One of the following two functions must be called to specify the integration tolerances for the backward problem. Note that this call must be made after the call to CVodeInitB or CVodeInitBS.

CVodeSStolerancesB

Call: `flag = CVodeSStolerancesB(cvode_mem, which, reltolB, abstolB);`

Description: The function CVodeSStolerancesB specifies scalar relative and absolute tolerances.

Arguments:

cvode_mem	(void *) pointer to the CVODES memory block returned by CVodeCreate.
which	(int) represents the identifier of the backward problem.
reltolB	(realtype) is the scalar relative error tolerance.
abstolB	(realtype) is the scalar absolute error tolerance.

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

CV_SUCCESS	The call to CVodeSStolerancesB was successful.
CV_MEM_NULL	The CVODES memory block was not initialized through a previous call to CVodeCreate.
CV_NO_MALLOC	The allocation function CVodeInit has not been called.
CV_NO_ADJ	The function CVodeAdjInit has not been previously called.
CV_ILL_INPUT	One of the input tolerances was negative.

CVodeSVtolerancesB

Call flag = CVodeSVtolerancesB(cvode_mem, which, reltolB, abstolB);

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

Arguments
- cvode_mem (void *) pointer to the CVODES memory block returned by CVodeCreate.
- which (int) represents the identifier of the backward problem.
- reltol (realtype) is the scalar relative error tolerance.
- abstol (N_Vector) is the vector of absolute error tolerances.

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

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

6.2.5 Linear solver initialization functions for backward problem

All linear solver modules in CVODES available for forward problems provide additional specification functions for backward problems. The initialization functions described in §4.5.3 cannot be directly used since the optional user-defined Jacobian-related functions have different prototypes for the backward problem than for the forward problem (see §6.3).

The following wrapper functions can be used to initialize one of the linear solver modules for the backward problem. Their arguments are identical to those of the functions in §4.5.3 with the exception of the additional second argument, which, the identifier of the backward problem.

```
flag = CVDenseB(cvode_mem, which, nB);
flag = CVBandB(cvode_mem, which, nB, mupperB, mlowerB);
flag = CVLapackDenseB(cvode_mem, which, nB);
flag = CVLapackBandB(cvode_mem, which, nB, mupperB, mlowerB);
flag = CVDiagB(cvode_mem, which);
flag = CVSuperLUMTB(cvode_mem, num_threads, which, nB, nnzB);
flag = CVKLUB(cvode_mem, which, nB, nnzB);
flag = CVSpgmrB(cvode_mem, which, pretypeB, maxlB);
flag = CVSpbcgB(cvode_mem, which, pretypeB, maxlB);
flag = CVSptfqmrB(cvode_mem, which, pretypeB, maxlB);
```

Their return value flag (of type int) can have any of the return values of their counterparts. If the cvode_mem argument was NULL, flag will be CVDLS_MEM_NULL, CVSLS_MEM_NULL, CVDIAG_MEM_NULL, or CVSPILS_MEM_NULL. Also, if which is not a valid identifier, the functions will return CVDLS_ILL_INPUT, CVSLS_ILL_INPUT, CVDIAG_ILL_INPUT, or CVSPILS_ILL_INPUT.

6.2.6 Backward integration function

The function CVodeB performs the integration of the backward problem. It is essentially a wrapper for the CVODES main integration function CVode and, in the case in which checkpoints were needed, it evolves the solution of the backward problem through a sequence of forward-backward integration pairs between consecutive checkpoints. The first run of each pair integrates the original IVP forward in time and stores interpolation data; the second run integrates the backward problem backward in time and performs the required interpolation to provide the solution of the IVP to the backward problem.

The function `CVodeB` does not return the solution `yB` itself. To obtain that, call the function `CVodeGetB`, which is also described below.

The call to `CVodeB` has the form

`CVodeB`

Call `flag = CVodeB(cvode_mem, tBout, itaskB);`

Description The function `CVodeB` integrates the backward ODE problem.

Arguments

- `cvode_mem` (`void *`) pointer to the CVODES memory returned by `CVodeCreate`.
- `tBout` (`realtype`) the next time at which a computed solution is desired.
- `itaskB` (`int`) a flag indicating the job of the solver for the next step. The `CV_NORMAL` task is to have the solver take internal steps until it has reached or just passed the user-specified value `tBout`. The solver then interpolates in order to return an approximate value of $yB($`tBout`$)$. The `CV_ONE_STEP` option tells the solver to take just one internal step in the direction of `tBout` and return.

Return value The return value `flag` (of type `int`) will be one of the following. For more details see §4.5.5.

`CV_SUCCESS`	`CVodeB` succeeded.
`CV_MEM_NULL`	`cvode_mem` was `NULL`.
`CV_NO_ADJ`	The function `CVodeAdjInit` has not been previously called.
`CV_NO_BCK`	No backward problem has been added to the list of backward problems by a call to `CVodeCreateB`
`CV_NO_FWD`	The function `CVodeF` has not been previously called.
`CV_ILL_INPUT`	One of the inputs to `CVodeB` is illegal.
`CV_BAD_ITASK`	The `itaskB` argument has an illegal value.
`CV_TOO_MUCH_WORK`	The solver took `mxstep` internal steps but could not reach `tBout`.
`CV_TOO_MUCH_ACC`	The solver could not satisfy the accuracy demanded by the user for some internal step.
`CV_ERR_FAILURE`	Error test failures occurred too many times during one internal time step.
`CV_CONV_FAILURE`	Convergence test failures occurred too many times during one internal time step.
`CV_LSETUP_FAIL`	The linear solver's setup function failed in an unrecoverable manner.
`CV_SOLVE_FAIL`	The linear solver's solve function failed in an unrecoverable manner.
`CV_BCKMEM_NULL`	The solver memory for the backward problem was not created with a call to `CVodeCreateB`.
`CV_BAD_TBOUT`	The desired output time `tBout` is outside the interval over which the forward problem was solved.
`CV_REIFWD_FAIL`	Reinitialization of the forward problem failed at the first checkpoint (corresponding to the initial time of the forward problem).
`CV_FWD_FAIL`	An error occurred during the integration of the forward problem.

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

In the case of multiple checkpoints and multiple backward problems, a given call to `CVodeB` in `CV_ONE_STEP` mode may not advance every problem one step, depending on the relative locations of the current times reached. But repeated calls will eventually advance all problems to `tBout`.

To obtain the solution `yB` to the backward problem, call the function `CVodeGetB` as follows:

`CVodeGetB`

Call `flag = CVodeGetB(cvode_mem, which, &tret, yB);`

Description The function `CVodeGetB` provides the solution `yB` of the backward ODE problem.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory returned by `CVodeCreate`.
- `which` (`int`) the identifier of the backward problem.
- `tret` (`realtype`) the time reached by the solver (output).
- `yB` (`N_Vector`) the backward solution at time `tret`.

Return value The return value `flag` (of type `int`) will be one of the following.
- `CV_SUCCESS` `CVodeGetB` was successful.
- `CV_MEM_NULL` `cvode_mem` is `NULL`.
- `CV_NO_ADJ` The function `CVodeAdjInit` has not been previously called.
- `CV_ILL_INPUT` The parameter `which` is an invalid identifier.

Notes The user must allocate space for `yB`.

6.2.7 Adjoint sensitivity optional input

At any time during the integration of the forward problem, the user can disable the checkpointing of the forward sensitivities by calling the following function:

`CVodeAdjSetNoSensi`

Call `flag = CVodeAdjSetNoSensi(cvode_mem);`

Description The function `CVodeAdjSetNoSensi` instructs `CVodeF` not to save checkpointing data for forward sensitivities anymore.

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

Return value The return value `flag` (of type `int`) is one of:
- `CV_SUCCESS` The call to `CVodeCreateB` was successful.
- `CV_MEM_NULL` `cvode_mem` was `NULL`.
- `CV_NO_ADJ` The function `CVodeAdjInit` has not been previously called.

6.2.8 Optional input functions for the backward problem

6.2.8.1 Main solver optional input functions

The adjoint module in CVODES provides wrappers for most of the optional input functions defined in §4.5.6.1. The only difference is that the user must specify the identifier `which` of the backward problem within the list managed by CVODES.

The optional input functions defined for the backward problem are:

```
flag = CVodeSetUserDataB(cvode_mem, which, user_dataB);
flag = CVodeSetIterTypeB(cvode_mem, which, iterB);
flag = CVodeSetMaxOrdB(cvode_mem, which, maxordB);
flag = CVodeSetMaxNumStepsB(cvode_mem, which, mxstepsB);
flag = CVodeSetInitStepB(cvode_mem, which, hinB)
flag = CVodeSetMinStepB(cvode_mem, which, hminB);
flag = CVodeSetMaxStepB(cvode_mem, which, hmaxB);
flag = CVodeSetStabLimDetB(cvode_mem, which, stldetB);
```

Their return value `flag` (of type `int`) can have any of the return values of their counterparts, but it can also be `CV_NO_ADJ` if `CVodeAdjInit` has not been called, or `CV_ILL_INPUT` if `which` was an invalid identifier.

6.2.8.2 Dense linear solver

Optional inputs for the CVDENSE linear solver module can be set for the backward problem through the following two functions:

`CVDlsSetDenseJacFnB`

Call `flag = CVDlsSetDenseJacFnB(cvode_mem, which, jacB);`

Description The function `CVDlsSetDenseJacFnB` specifies the dense Jacobian approximation function to be used for the backward problem.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory returned by `CVodeCreate`.
- `which` (`int`) represents the identifier of the backward problem.
- `jacB` (`CVDlsDenseJacFnB`) user-defined dense Jacobian approximation function.

Return value The return value `flag` (of type `int`) is one of:
- `CVDLS_SUCCESS` `CVDlsSetDenseJacFnB` succeeded.
- `CVDLS_MEM_NULL` `cvode_mem` was `NULL`.
- `CVDLS_NO_ADJ` The function `CVodeAdjInit` has not been previously called.
- `CVDLS_LMEM_NULL` The linear solver has not been initialized with a call to `CVDenseB` or `CVLapackDenseB`.
- `CVDLS_ILL_INPUT` The parameter `which` represented an invalid identifier.

Notes The function type `CVDlsDenseJacFnB` is described in §6.3.5.

`CVDlsSetDenseJacFnBS`

Call `flag = CVDlsSetDenseJacFnBS(cvode_mem, which, jacBS);`

Description The function `CVDlsSetDenseJacFnBS` specifies the dense Jacobian approximation function to be used for the backward problem, in the case where the backward problem depends on the forward sensitivities.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory returned by `CVodeCreate`.
- `which` (`int`) represents the identifier of the backward problem.
- `jacBS` (`CVDlsDenseJacFnBS`) user-defined dense Jacobian approximation function.

Return value The return value `flag` (of type `int`) is one of:
- `CVDLS_SUCCESS` `CVDlsSetDenseJacFnBS` succeeded.
- `CVDLS_MEM_NULL` `cvode_mem` was `NULL`.
- `CVDLS_NO_ADJ` The function `CVodeAdjInit` has not been previously called.
- `CVDLS_LMEM_NULL` The linear solver has not been initialized with a call to `CVDenseB` or `CVLapackDenseB`.
- `CVDLS_ILL_INPUT` The parameter `which` represented an invalid identifier.

Notes The function type `CVDlsDenseJacFnBS` is described in §6.3.5.

6.2.8.3 Band linear solver

Optional inputs for the CVBAND linear solver module can be set for the backward problem through the following two functions:

`CVDlsSetBandJacFnB`

Call `flag = CVDlsSetBandJacFnB(cvode_mem, which, jacB);`

Description The function `CVDlsSetBandJacFnB` specifies the banded Jacobian approximation function to be used for the backward problem.

Arguments
: `cvode_mem` (`void *`) pointer to the CVODES memory returned by `CVodeCreate`.
: `which` (`int`) represents the identifier of the backward problem.
: `jacB` (`CVDlsBandJacFnB`) user-defined banded Jacobian approximation function.

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

CVDLS_SUCCESS	`CVDlsSetBandJacFnB` succeeded.
CVDLS_MEM_NULL	`cvode_mem` was NULL.
CVDLS_NO_ADJ	The function `CVodeAdjInit` has not been previously called.
CVDLS_LMEM_NULL	The linear solver has not been initialized with a call to `CVBandB` or `CVLapackBandB`.
CVDLS_ILL_INPUT	The parameter `which` represented an invalid identifier.

Notes The function type `CVDlsBandJacFnB` is described in §6.3.6.

`CVDlsSetBandJacFnBS`

Call `flag = CVDlsSetBandJacFnBS(cvode_mem, which, jacBS);`

Description The function `CVDlsSetBandJacFnBS` specifies the banded Jacobian approximation function to be used for the backward problem, in the case where the backward problem depends on the forward sensitivities.

Arguments
: `cvode_mem` (`void *`) pointer to the CVODES memory returned by `CVodeCreate`.
: `which` (`int`) represents the identifier of the backward problem.
: `jacBS` (`CVDlsBandJacFnBS`) user-defined banded Jacobian approximation function.

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

CVDLS_SUCCESS	`CVDlsSetBandJacFnBS` succeeded.
CVDLS_MEM_NULL	`cvode_mem` was NULL.
CVDLS_NO_ADJ	The function `CVodeAdjInit` has not been previously called.
CVDLS_LMEM_NULL	The linear solver has not been initialized with a call to `CVBandB` or `CVLapackBandB`.
CVDLS_ILL_INPUT	The parameter `which` represented an invalid identifier.

Notes The function type `CVDlsBandJacFnBS` is described in §6.3.6.

6.2.8.4 Sparse linear solvers

Optional inputs for the CVKLU and CVSUPERLUMT linear solver modules can be set for the backward problem through the following functions.

The following wrapper functions can be used to set the fill-reducing ordering and, in the case of KLU, reinitialize the sparse solver in the sparse linear solver modules for the backward problem. Their arguments are identical to those of the functions in §4.5.3 with the exception of the additional second argument, `which`, the identifier of the backward problem.

```
flag = CVKLUReInitB(cvode_mem, which, nB, nnzB, reinit_typeB);
flag = CVKLUSetOrderingB(cvode_mem, which, ordering_choiceB);
flag = CVSuperLUMTSetOrderingB(cvode_mem, which, ordering_choiceB);
```

Their return value `flag` (of type `int`) can have any of the return values of their counterparts. If the `cvode_mem` argument was `NULL`, `flag` will be `CVSLS_MEM_NULL`. Also, if `which` is not a valid identifier, the functions will return `CVSLS_ILL_INPUT`.

`CVSlsSetSparseJacFnB`

Call `flag = CVSlsSetSparseJacFnB(cvode_mem, which, jacB);`

Description The function `CVSlsSetSparseJacFnB` specifies the sparse Jacobian approximation function to be used for the backward problem.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory returned by `CVodeCreate`.
- `which` (`int`) represents the identifier of the backward problem.
- `jacB` (`CVSlsSparseJacFnB`) user-defined sparse Jacobian approximation function.

Return value The return value `flag` (of type `int`) is one of:
- `CVSLS_SUCCESS` `CVSlsSetSparseJacFnB` succeeded.
- `CVSLS_MEM_NULL` `cvode_mem` was `NULL`.
- `CVSLS_NO_ADJ` The function `CVodeAdjInit` has not been previously called.
- `CVSLS_LMEM_NULL` The linear solver has not been initialized with a call to `CVKLUB` or `CVSuperLUMTB`.
- `CVSLS_ILL_INPUT` The parameter `which` represented an invalid identifier.

Notes The function type `CVSlsSparseJacFnB` is described in §6.3.7.

`CVSlsSetSparseJacFnBS`

Call `flag = CVSlsSetSparseJacFnBS(cvode_mem, which, jacBS);`

Description The function `CVSlsSetSparseJacFnBS` specifies the sparse Jacobian approximation function to be used for the backward problem, in the case where the backward problem depends on the forward sensitivities.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory returned by `CVodeCreate`.
- `which` (`int`) represents the identifier of the backward problem.
- `jacBS` (`CVSlsSparseJacFnBS`) user-defined sparse Jacobian approximation function.

Return value The return value `flag` (of type `int`) is one of:
- `CVSLS_SUCCESS` `CVSlsSetSparseJacFnBS` succeeded.
- `CVSLS_MEM_NULL` `cvode_mem` was `NULL`.
- `CVSLS_NO_ADJ` The function `CVodeAdjInit` has not been previously called.
- `CVSLS_LMEM_NULL` The linear solver has not been initialized with a call to `CVKLUB` or `CVSuperLUMTB`.
- `CVSLS_ILL_INPUT` The parameter `which` represented an invalid identifier.

Notes The function type `CVSlsSparseJacFnBS` is described in §6.3.7.

6.2.8.5 SPILS linear solvers

Optional inputs for the CVSPILS linear solver module can be set for the backward problem through the following functions:

`CVSpilsSetPreconditionerB`

Call `flag = CVSpilsSetPreconditionerB(cvode_mem, which, psetupB, psolveB);`

Description The function `CVSpilsSetPrecSolveFnB` specifies the preconditioner setup and solve functions for the backward integration.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `which` (`int`) the identifier of the backward problem.
- `psetupB` (`CVSpilsPrecSetupFnB`) user-defined preconditioner setup function.

psolveB (CVSpilsPrecSolveFnB) user-defined preconditioner solve function.

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

CVSPILS_SUCCESS	The optional value has been successfully set.
CVSPILS_MEM_NULL	cvode_mem was NULL.
CVSPILS_LMEM_NULL	The CVSPILS linear solver has not been initialized.
CVSPILS_NO_ADJ	The function CVodeAdjInit has not been previously called.
CVSPILS_ILL_INPUT	The parameter which represented an invalid identifier.

Notes The function types CVSpilsPrecSolveFnB and CVSpilsPrecSetupFnB are described in §6.3.9 and §6.3.10, resp. The psetupB argument may be NULL if no setup operation is involved in the preconditioner.

CVSpilsSetPreconditionerBS

Call `flag = CVSpilsSetPreconditionerBS(cvode_mem, which, psetupBS, psolveBS);`

Description The function CVSpilsSetPrecSolveFnBS specifies the preconditioner setup and solve functions for the backward integration, in the case where the backward problem depends on the forward sensitivities.

Arguments
cvode_mem (void *) pointer to the CVODES memory block.
which (int) the identifier of the backward problem.
psetupBS (CVSpilsPrecSetupFnBS) user-defined preconditioner setup function.
psolveBS (CVSpilsPrecSolveFnBS) user-defined preconditioner solve function.

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

CVSPILS_SUCCESS	The optional value has been successfully set.
CVSPILS_MEM_NULL	cvode_mem was NULL.
CVSPILS_LMEM_NULL	The CVSPILS linear solver has not been initialized.
CVSPILS_NO_ADJ	The function CVodeAdjInit has not been previously called.
CVSPILS_ILL_INPUT	The parameter which represented an invalid identifier.

Notes The function types CVSpilsPrecSolveFnBS and CVSpilsPrecSetupFnBS are described in §6.3.9 and §6.3.10, resp. The psetupBS argument may be NULL if no setup operation is involved in the preconditioner.

CVSpilsSetJacTimesVecFnB

Call `flag = CVSpilsSetJacTimesVecFnB(cvode_mem, which, jtvB);`

Description The function CVSpilsSetJacTimesFnB specifies the Jacobian-vector product function to be used.

Arguments
cvode_mem (void *) pointer to the CVODES memory block.
which (int) the identifier of the backward problem.
jtvB (CVSpilsJacTimesVecFnB) user-defined Jacobian-vector product function.

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

CVSPILS_SUCCESS	The optional value has been successfully set.
CVSPILS_MEM_NULL	cvode_mem was NULL.
CVSPILS_LMEM_NULL	The CVSPILS linear solver has not been initialized.
CVSPILS_NO_ADJ	The function CVodeAdjInit has not been previously called.
CVSPILS_ILL_INPUT	The parameter which represented an invalid identifier.

Notes The function type CVSpilsJacTimesVecFnB is described in §6.3.8.

`CVSpilsSetJacTimesVecFnBS`

Call `flag = CVSpilsSetJacTimesVecFnBS(cvode_mem, which, jtvBS);`

Description The function `CVSpilsSetJacTimesFnBS` specifies the Jacobian-vector product function to be used, in the case where the backward problem depends on the forward sensitivities.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `which` (`int`) the identifier of the backward problem.
- `jtvBS` (`CVSpilsJacTimesVecFnBS`) user-defined Jacobian-vector product function.

Return value The return value `flag` (of type `int`) is one of:
- `CVSPILS_SUCCESS` The optional value has been successfully set.
- `CVSPILS_MEM_NULL` `cvode_mem` was `NULL`.
- `CVSPILS_LMEM_NULL` The CVSPILS linear solver has not been initialized.
- `CVSPILS_NO_ADJ` The function `CVodeAdjInit` has not been previously called.
- `CVSPILS_ILL_INPUT` The parameter `which` represented an invalid identifier.

Notes The function type `CVSpilsJacTimesVecFnBS` is described in §6.3.8.

`CVSpilsSetGSTypeB`

Call `flag = CVSpilsSetGSType(cvode_mem, which, gstypeB);`

Description The function `CVSpilsSetGSTypeB` specifies the type of Gram-Schmidt orthogonalization to be used with CVSPGMR. This must be 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 CVODES memory block.
- `which` (`int`) the identifier of the backward problem.
- `gstypeB` (`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` `cvode_mem` was `NULL`.
- `CVSPILS_LMEM_NULL` The CVSPILS linear solver has not been initialized.
- `CVSPILS_NO_ADJ` The function `CVodeAdjInit` has not been previously called.
- `CVSPILS_ILL_INPUT` The parameter `which` represented an invalid identifier, or the value of `gstypeB` was not valid.

Notes The default value is `MODIFIED_GS`.

This option is available only with CVSPGMR.

`CVSpilsSetMaxlB`

Call `flag = CVSpilsSetMaxlB(cvode_mem, which, maxlB);`

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `which` (`int`) the identifier of the backward problem.
- `maxlB` (`realtype`) 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` `cvode_mem` was `NULL`.

CVSPILS_LMEM_NULL	The CVSPILS linear solver has not been initialized.
CVSPILS_NO_ADJ	The function CVodeAdjInit has not been previously called.
CVSPILS_ILL_INPUT	The parameter which represented an invalid identifier.

Notes The maximum subspace dimension is initially specified in the call to CVodeSpbcgB or CVodeSptfqmrB. The call to CVodeSpilsSetMaxlB is needed only if maxlB is being changed from its previous value.

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

CVSpilsSetEpsLinB

Call `flag = CVSpilsSetEpsLinB(cvode_mem, which, eplifacB);`

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

Arguments

cvode_mem	(void *) pointer to the CVODES memory block.
which	(int) the identifier of the backward problem.
eplifacB	(realtype) value of the convergence test constant reduction factor (≥ 0.0).

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

CVSPILS_SUCCESS	The optional value has been successfully set.
CVSPILS_MEM_NULL	cvode_mem was NULL.
CVSPILS_LMEM_NULL	The CVSPILS linear solver has not been initialized.
CVSPILS_NO_ADJ	The function CVodeAdjInit has not been previously called.
CVSPILS_ILL_INPUT	The parameter which represented an invalid identifier, or eplifacB was negative.

Notes The default value is 0.05. Passing a value eplifacB= 0.0 also indicates using the default value.

CVSpilsSetPrecTypeB

Call `flag = CVSpilsSetPrecTypeB(cvode_mem, which, pretypeB);`

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

Arguments

cvode_mem	(void *) pointer to the CVODES memory block.
which	(int) the identifier of the backward problem.
pretypeB	(int) specifies the type of prconditioning 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	cvode_mem was NULL.
CVSPILS_LMEM_NULL	The CVSPILS linear solver has not been initialized.
CVSPILS_NO_ADJ	The function CVodeAdjInit has not been previously called.
CVSPILS_ILL_INPUT	The parameter which represented an invalid identifier, or the value of pretypeB was not valid.

Notes The preconditioning type is initially specified in the call to the linear solver specification function (see §6.2.5). The call to CVSpilsSetPrecTypeB is needed only if pretypeB is being changed from its previous value.

6.2.9 Optional output functions for the backward problem

The user of the adjoint module in CVODES has access to any of the optional output functions described in §4.5.8, both for the main solver and for the linear solver modules. The first argument of these `CVodeGet*` and `CVode*Get*` functions is the pointer to the CVODES memory block for the backward problem. In order to call any of these functions, the user must first call the following function to obtain this pointer.

CVodeGetAdjCVodeBm]TJ /R409 9.96264 Tf 8

Ta952 Td [(o)0.431299(l)0.441892(cvodes)442326]TJ /R409 9018364 Tf 135 memory block ...

`CVodeQuadInitB`

Call `flag = CVodeQuadInitB(cvode_mem, which, rhsQB, yQB0);`

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `which` (`int`) the identifier of the backward problem.
- `rhsQB` (`CVQuadRhsFnB`) is the C function which computes fQB, the right-hand side of the backward quadrature equations. This function has the form `rhsQB(t, y, yB, qBdot, user_dataB)` (see §6.3.3).
- `yQB0` (`N_Vector`) is the value of the quadrature variables at `tB0`.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeQuadInitB` was successful.
- `CV_MEM_NULL` `cvode_mem` was `NULL`.
- `CV_NO_ADJ` The function `CVodeAdjInit` has not been previously called.
- `CV_MEM_FAIL` A memory allocation request has failed.
- `CV_ILL_INPUT` The parameter `which` is an invalid identifier.

The function `CVodeQuadInitBS` initializes and allocates memory for the backward integration of quadrature equations that depends on the forward sensitivities.

`CVodeQuadInitBS`

Call `flag = CVodeQuadInitBS(cvode_mem, which, rhsQBS, yQBS0);`

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `which` (`int`) the identifier of the backward problem.
- `rhsQBS` (`CVQuadRhsFnBS`) is the C function which computes $fQBS$, the right-hand side of the backward quadrature equations. This function has the form `rhsQBS(t, y, yS, yB, qBdot, user_dataB)` (see §6.3.4).
- `yQBS0` (`N_Vector`) is the value of the sensitivity-dependent quadrature variables at `tB0`.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeQuadInitBS` was successful.
- `CV_MEM_NULL` `cvode_mem` was `NULL`.
- `CV_NO_ADJ` The function `CVodeAdjInit` has not been previously called.
- `CV_MEM_FAIL` A memory allocation request has failed.
- `CV_ILL_INPUT` The parameter `which` is an invalid identifier.

The integration of quadrature equations during the backward phase can be re-initialized by calling

`CVodeQuadReInitB`

Call `flag = CVodeQuadReInitB(cvode_mem, which, yQB0);`

Description The function `CVodeQuadReInitB` re-initializes the backward quadrature integration.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `which` (`int`) the identifier of the backward problem.
- `yQB0` (`N_Vector`) is the value of the quadrature variables at `tB0`.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeQuadReInitB` was successful.

CV_MEM_NULL	cvode_mem was NULL.
CV_NO_ADJ	The function CVodeAdjInit has not been previously called.
CV_MEM_FAIL	A memory allocation request has failed.
CV_NO_QUAD	Quadrature integration was not activated through a previous call to CVodeQuadInitB.
CV_ILL_INPUT	The parameter which is an invalid identifier.

Notes The function CVodeQuadReInitB can be called after a call to either CVodeQuadInitB or CVodeQuadInitBS.

6.2.10.2 Backward quadrature extraction function

To extract the values of the quadrature variables at the last return time of CVodeB, CVODES provides a wrapper for the function CVodeGetQuad (see §4.7.3). The call to this function has the form

CVodeGetQuadB

Call `flag = CVodeGetQuadB(cvode_mem, which, &tret, yQB);`

Description The function CVodeGetQuadB returns the quadrature solution vector after a successful return from CVodeB.

Arguments

cvode_mem	(void *) pointer to the CVODES memory.
tret	(realtype) the time reached by the solver (output).
yQB	(N_Vector) the computed quadrature vector.

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

CV_SUCCESS	CVodeGetQuadB was successful.
CV_MEM_NULL	cvode_mem is NULL.
CV_NO_ADJ	The function CVodeAdjInit has not been previously called.
CV_NO_QUAD	Quadrature integration was not initialized.
CV_BAD_DKY	yQB was NULL.
CV_ILL_INPUT	The parameter which is an invalid identifier.

6.2.10.3 Optional input/output functions for backward quadrature integration

Optional values controlling the backward integration of quadrature equations can be changed from their default values through calls to one of the following functions which are wrappers for the corresponding optional input functions defined in §4.7.4. The user must specify the identifier which of the backward problem for which the optional values are specified.

```
flag = CVodeSetQuadErrConB(cvode_mem, which, errconQ);
flag = CVodeQuadSStolerancesB(cvode_mem, which, reltolQ, abstolQ);
flag = CVodeQuadSVtolerancesB(cvode_mem, which, reltolQ, abstolQ);
```

Their return value flag (of type int) can have any of the return values of its counterparts, but it can also be CV_NO_ADJ if the function CVodeAdjInit has not been previously called or CV_ILL_INPUT if the parameter which was an invalid identifier.

Access to optional outputs related to backward quadrature integration can be obtained by calling the corresponding CVodeGetQuad* functions (see §4.7.5). A pointer cvode_memB to the CVODES memory block for the backward problem, required as the first argument of these functions, can be obtained through a call to the functions CVodeGetAdjCVodeBmem (see §6.2.9).

6.3 User-supplied functions for adjoint sensitivity analysis

In addition to the required ODE right-hand side function and any optional functions for the forward

Purpose
This function evaluates the right-hand side $f_B(t, y, y_B, s)$ of the backward problem ODE system. This could be either (2.19) or (2.22).

Arguments
- `t` is the current value of the independent variable.
- `y` is the current value of the forward solution vector.
- `yS` a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution.
- `yB` is the current value of the backward dependent variable vector.
- `yBdot` is the output vector containing the right-hand side f_B of the backward ODE problem.
- `user_dataB` is a pointer to user data, same as passed to `CVodeSetUserDataB`.

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

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

The `y`, `yB`, and `yBdot` arguments are all of type `N_Vector`, but `yB` and `yBdot` typically have different internal representations from `y`. Likewise for each `yS[i]`. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with CVODES do not perform any consistency checks with respect to their `N_Vector` arguments (see §7.1 and §7.2).

The `user_dataB` pointer is passed to the user's `rhsBS` function every time it is called and can be the same as the `user_data` pointer used for the forward problem.

Before calling the user's `rhsBS` function, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the right-hand side function which will halt the integration and `CVodeB` will return `CV_RHSFUNC_FAIL`.

6.3.3 Quadrature right-hand side for the backward problem

The user must provide an `fQB` function of type `CVQuadRhsFnB` defined by

`CVQuadRhsFnB`

Definition
```
typedef int (*CVQuadRhsFnB)(realtype t, N_Vector y, N_Vector yB,
                            N_Vector qBdot, void *user_dataB);
```

Purpose
This function computes the quadrature equation right-hand side for the backward problem.

Arguments
- `t` is the current value of the independent variable.
- `y` is the current value of the forward solution vector.
- `yB` is the current value of the backward dependent variable vector.
- `qBdot` is the output vector containing the right-hand side `fQB` of the backward quadrature equations.
- `user_dataB` is a pointer to user data, same as passed to `CVodeSetUserDataB`.

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

Notes Allocation of memory for rhsvalBQ is handled within CVODES.

The y, yB, and qBdot arguments are all of type N_Vector, but they typically do not all have the same representation. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with CVODES do not perform any consistency checks with repsect to their N_Vector arguments (see §7.1 and §7.2).

The user_dataB pointer is passed to the user's fQB function every time it is called and can be the same as the user_data pointer used for the forward problem.

Before calling the user's fQB function, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the quadrature right-hand side function which will halt the integration and CVodeB will return CV_QRHSFUNC_FAIL.

6.3.4 Sensitivity-dependent quadrature right-hand side for the backward problem

The user must provide an fQBS function of type CVQuadRhsFnBS defined by

CVQuadRhsFnBS

Definition
```
typedef int (*CVQuadRhsFnBS)(realtype t, N_Vector y, N_Vector *yS,
                             N_Vector yB, N_Vector qBdot,
                             void *user_dataB);
```

Purpose This function computes the quadrature equation right-hand side for the backward problem.

Arguments
- t is the current value of the independent variable.
- y is the current value of the forward solution vector.
- yS a pointer to an array of Ns vectors containing the sensitivities of the forward solution.
- yB is the current value of the backward dependent variable vector.
- qBdot is the output vector containing the right-hand side fQBS of the backward quadrature equations.
- user_dataB is a pointer to user data, same as passed to CVodeSetUserDataB.

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

Notes Allocation of memory for qBdot is handled within CVODES.

The y, yS, and qBdot arguments are all of type N_Vector, but they typically do not all have the same internal representation. Likewise for each yS[i]. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with CVODES do not perform any consistency checks with repsect to their N_Vector arguments (see §7.1 and §7.2).

The user_dataB pointer is passed to the user's fQBS function every time it is called and can be the same as the user_data pointer used for the forward problem.

Before calling the user's fQBS function, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the quadrature right-hand side function which will halt the integration and CVodeB will return CV_QRHSFUNC_FAIL.

6.3.5 Jacobian information for the backward problem (direct method with dense Jacobian)

If the direct linear solver with dense treatment of the Jacobian is selected for the backward problem (i.e. CVDenseB or CVLapackDenseB is called in step 21 of §6.1), the user may provide, through a call to CVDlsSetDenseJacFnB or CVDlsSetDenseJacFnBS (see §6.2.8), a function of one of the following two types:

CVDlsDenseJacFnB

Definition
```
typedef int (*CVDlsDenseJacFnB)(long int NeqB, realtype t, N_Vector y,
                                N_Vector yB, N_Vector fyB,
                                DlsMat JacB, void *user_dataB,
                                N_Vector tmp1B, N_Vector tmp2B,
                                N_Vector tmp3B);
```

Purpose
This function computes the dense Jacobian of the backward problem (or an approximation to it).

Arguments
- NeqB is the backward problem size (number of equations).
- t is the current value of the independent variable.
- y is the current value of the forward solution vector.
- yB is the current value of the backward dependent variable vector.
- fyB is the current value of the backward right-hand side function f_B.
- JacB is the output approximate dense Jacobian matrix.
- user_dataB is a pointer to user data – the same as passed to CVodeSetUserDataB.
- tmp1B
- tmp2B
- tmp3B are pointers to memory allocated for variables of type N_Vector which can be used by CVDlsDenseJacFnB as temporary storage or work space.

Return value
A CVDlsDenseJacFnB should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES 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, CVodeB returns CV_LSETUP_FAIL and CVDENSE sets last_flag to CVDLS_JACFUNC_UNRECVR).

Notes
A user-supplied dense Jacobian function must load the NeqB by NeqB dense matrix JacB with an approximation to the Jacobian matrix at the point (t,y,yB), where y is the solution of the original IVP at time tt and yB is the solution of the backward problem at the same time. Only nonzero elements need to be loaded into JacB as this matrix is set to zero before the call to the Jacobian function. The type of JacB is DlsMat. The user is referred to §4.6.5 for details regarding accessing a DlsMat object.

Before calling the user's CVDlsDenseJacFnB, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the Jacobian function which will halt the integration (CVodeB returns CV_LSETUP_FAIL and CVDENSE sets last_flag to CVDLS_JACFUNC_UNRECVR).

CVDlsDenseJacFnBS

Definition
```
typedef int (*CVDlsDenseJacFnBS)(long int NeqB, realtype t, N_Vector y,
                                 N_Vector *yS, N_Vector yB, N_Vector fyB,
                                 DlsMat JacB, void *user_dataB,
                                 N_Vector tmp1B, N_Vector tmp2B,
                                 N_Vector tmp3B);
```

Purpose This function computes the dense Jacobian of the backward problem (or an approximation to it), in the case where the backward problem depends on the forward sensitivities.

Arguments
- `NeqB` is the backward problem size (number of equations).
- `t` is the current value of the independent variable.
- `y` is the current value of the forward solution vector.
- `yS` a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution.
- `yB` is the current value of the backward dependent variable vector.
- `fyB` is the current value of the backward right-hand side function f_B.
- `JacB` is the output approximate dense Jacobian matrix.
- `user_dataB` is a pointer to user data – the same as passed to `CVodeSetUserDataB`.
- `tmp1B`
- `tmp2B`
- `tmp3B` are pointers to memory allocated for variables of type `N_Vector` which can be used by `CVDlsDenseJacFnBS` as temporary storage or work space.

Return value A `CVDlsDenseJacFnBS` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES 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, `CVodeB` returns `CV_LSETUP_FAIL` and CVDENSE sets `last_flag` to `CVDLS_JACFUNC_UNRECVR`).

Notes A user-supplied dense Jacobian function must load the `NeqB` by `NeqB` dense matrix `JacB` with an approximation to the Jacobian matrix at the point (`t`,`y`,`yS`,`yB`), where `y` is the solution of the original IVP at time `tt`, `yS` is the vector of forward sensitivities at time `tt`, and `yB` is the solution of the backward problem at the same time. Only nonzero elements need to be loaded into `JacB` as this matrix is set to zero before the call to the Jacobian function. The type of `JacB` is `DlsMat`. The user is referred to §4.6.5 for details regarding accessing a `DlsMat` object.

Before calling the user's `CVDlsDenseJacFnBS`, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the Jacobian function which will halt the integration (`CVodeB` returns `CV_LSETUP_FAIL` and CVDENSE sets `last_flag` to `CVDLS_JACFUNC_UNRECVR`).

6.3.6 Jacobian information for the backward problem (direct method with banded Jacobian)

If the direct linear solver with banded treatment of the Jacobian is selected for the backward problem (i.e. `CVBandB` or `CVLapackBandB` is called in step 21 of §6.1), the user may provide, through a call to `CVDlsSetBandJacFnB` or `CVDlsSetBandJacFnBS` (see §6.2.8), a function of one of the following two types:

`CVDlsBandJacFnB`

Definition
```
typedef int (*CVDlsBandJacFnB)(long int NeqB,
                                long int mupperB, long int mlowerB,
                                realtype t, N_Vector y,
                                N_Vector yB, N_Vector fyB,
                                DlsMat JacB, void *user_dataB,
                                N_Vector tmp1B, N_Vector tmp2B,
                                N_Vector tmp3B);
```

Purpose This function computes the banded Jacobian of the backward problem (or a banded approximation to it).

Arguments
- `NeqB` is the backward problem size.
- `mlowerB`
- `mupperB` are the lower and upper half-bandwidth of the Jacobian.
- `t` is the current value of the independent variable.
- `y` is the current value of the forward solution vector.
- `yB` is the current value of the backward dependent variable vector.
- `fyB` is the current value of the backward right-hand side function f_B.
- `JacB` is the output approximate band Jacobian matrix.
- `user_dataB` is a pointer to user data – the same as passed to `CVodeSetUserDataB`.
- `tmp1B`
- `tmp2B`
- `tmp3B` are pointers to memory allocated for variables of type `N_Vector` which can be used by `CVDlsBandJacFnB` as temporary storage or work space.

Return value A `CVDlsBandJacFnB` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES 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, `CVodeB` returns `CV_LSETUP_FAIL` and CVDENSE sets `last_flag` to `CVDLS_JACFUNC_UNRECVR`).

Notes A user-supplied band Jacobian function must load the band matrix `JacB` (of type `DlsMat`) with the elements of the Jacobian at the point (`t`,`y`,`yB`), where `y` is the solution of the original IVP at time `tt` and `yB` is the solution of the backward problem at the same time. Only nonzero elements need to be loaded into `JacB` because `JacB` is preset to zero before the call to the Jacobian function. More details on the accessor macros provided for a `DlsMat` object and on the rest of the arguments passed to a function of type `CVDlsBandJacFnB` are given in §4.6.6.

!

Before calling the user's `CVDlsBandJacFnB`, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the Jacobian function which will halt the integration (`CVodeB` returns `CV_LSETUP_FAIL` and CVBAND sets `last_flag` to `CVDLS_JACFUNC_UNRECVR`).

`CVDlsBandJacFnBS`

Definition
```
typedef int (*CVDlsBandJacFnBS)(long int NeqB,
                                 long int mupperB, long int mlowerB,
                                 realtype t, N_Vector y, N_Vector *yS,
                                 N_Vector yB, N_Vector fyB,
                                 DlsMat JacB, void *user_dataB,
                                 N_Vector tmp1B, N_Vector tmp2B,
                                 N_Vector tmp3B);
```

Purpose This function computes the banded Jacobian of the backward problem (or a banded approximation to it), in the case where the backward problem depends on the forward sensitivities.

Arguments
- `NeqB` is the backward problem size.
- `mlowerB`
- `mupperB` are the lower and upper half-bandwidth of the Jacobian.
- `t` is the current value of the independent variable.
- `y` is the current value of the forward solution vector.
- `yS` a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution.
- `yB` is the current value of the backward dependent variable vector.
- `fyB` is the current value of the backward right-hand side function f_B.
- `JacB` is the output approximate band Jacobian matrix.
- `user_dataB` is a pointer to user data – the same as passed to `CVodeSetUserDataB`.
- `tmp1B`
- `tmp2B`
- `tmp3B` are pointers to memory allocated for variables of type `N_Vector` which can be used by `CVDlsBandJacFnB` as temporary storage or work space.

Return value A `CVDlsBandJacFnBS` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES 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, `CVodeB` returns `CV_LSETUP_FAIL` and CVDENSE sets `last_flag` to `CVDLS_JACFUNC_UNRECVR`).

Notes A user-supplied band Jacobian function must load the band matrix `JacB` (of type `DlsMat`) with the elements of the Jacobian at the point (t,y,yS,yB), where `y` is the solution of the original IVP at time `tt`, `yS` is the vector of forward sensitivities at time `tt`, and `yB` is the solution of the backward problem at the same time. Only nonzero elements need to be loaded into `JacB` because `JacB` is preset to zero before the call to the Jacobian function. More details on the accessor macros provided for a `DlsMat` object and on the rest of the arguments passed to a function of type `CVDlsBandJacFnBS` are given in §4.6.6.

Before calling the user's `CVDlsBandJacFnBS`, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the Jacobian function which will halt the integration (`CVodeB` returns `CV_LSETUP_FAIL` and CVBAND sets `last_flag` to `CVDLS_JACFUNC_UNRECVR`).

6.3.7 Jacobian information for the backward problem (direct method with sparse Jacobian)

If the direct linear solver with sparse treatment of the Jacobian is selected for the backward problem (i.e. `CVKLUB` or `CVSuperLUMTB` is called in step 21 of §6.1), the user must provide, through a call to `CVSlsSetSparseJacFnB` or `CVSlsSetSparseJacFnBS` (see §6.2.8), a function of one of the following two types:

`CVSlsSparseJacFnB`

Definition
```
typedef int (*CVSlsSparseJacFnB)(realtype t, N_Vector y,
                                 N_Vector yB, N_Vector fyB,
                                 SlsMat JacB, void *user_dataB,
                                 N_Vector tmp1B, N_Vector tmp2B,
                                 N_Vector tmp3B);
```

Purpose This function computes the sparse Jacobian of the backward problem (or an approximation to it).

Arguments

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

`y` is the current value of the forward solution vector.

`yB` is the current value of the backward dependent variable vector.

`fyB` is the current value of the backward right-hand side function f_B.

`JacB` is the output approximate sparse Jacobian matrix.

`user_dataB` is a pointer to user data – the same as passed to `CVodeSetUserDataB`.

`tmp1B`

`tmp2B`

`tmp3B` are pointers to memory allocated for variables of type `N_Vector` which can be used by `CVSlsSparseJacFnB` as temporary storage or work space.

Return value A `CVSlsSparseJacFnB` should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES 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, `CVodeB` 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 `JacB` with the elements of the Jacobian at the point (`t`,`y`,`yB`), where `y` is the solution of the original IVP at time `tt` and `yB` is the solution of the backward problem at the same time. Storage for `JacB` already exists on entry to this function, although the user should ensure that sufficient space is allocated in `JacB` 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 `JacB` 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 §9.2. More details on the rest of the arguments passed to a function of type `CVSlsSparseJacFnB` are given in §4.6.7.

Before calling the user's `CVSlsSparseJacFnB`, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the Jacobian function which will halt the integration (`CVodeB` returns `CV_LSETUP_FAIL` and CVKLU or CVSUPERLUMT sets `last_flag` to `CVSLS_JACFUNC_UNRECVR`).

`CVSlsSparseJacFnBS`

Definition

```
typedef int (*CVSlsSparseJacFnBS)(realtype t, N_Vector y, N_Vector *yS,
                                  N_Vector yB, N_Vector fyB,
                                  SlsMat JacB, void *user_dataB,
                                  N_Vector tmp1B, N_Vector tmp2B,
                                  N_Vector tmp3B);
```

Purpose This function computes the sparse Jacobian of the backward problem (or an approximation to it), in the case where the backward problem depends on the forward sensitivities.

Arguments

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

`y` is the current value of the forward solution vector.

`yS` a pointer to an array of `Ns` vectors containing the sensitivities of the forward solution.

`yB` is the current value of the backward dependent variable vector.

`fyB` is the current value of the backward right-hand side function f_B.

`JacB` is the output approximate sparse Jacobian matrix.

`user_dataB` is a pointer to user data – the same as passed to `CVodeSetUserDataB`.

`tmp1B`

`tmp2B`

tmp3B are pointers to memory allocated for variables of type N_Vector which can be used by CVSlsSparseJacFnB as temporary storage or work space.

Return value A CVSlsSparseJacFnBS should return 0 if successful, a positive value if a recoverable error occurred (in which case CVODES 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, CVodeB 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 JacB with the elements of the Jacobian at the point (t,y,yS,yB), where y is the solution of the original IVP at time tt, yS is the vector of forward sensitivities at time tt, and yB is the solution of the backward problem at the same time. Storage for JacB already exists on entry to this function, although the user should ensure that sufficient space is allocated in JacB 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 JacB 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 §9.2. More details on the rest of the arguments passed to a function of type CVSlsSparseJacFnBS are given in §4.6.7.

Before calling the user's CVSlsSparseJacFnBS, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the Jacobian function which will halt the integration (CVodeB returns CV_LSETUP_FAIL and CVKLU or CVSUPERLUMT sets last_flag to CVSLS_JACFUNC_UNRECVR).

6.3.8 Jacobian information for the backward problem (matrix-vector product)

If one of the Krylov iterative linear solvers SPGMR, SPBCG, or SPTFQMR is selected (CVodeSp*B is called in step 21 of §6.1), the user may provide a function of one of the following two types:

CVSpilsJacTimesVecFnB

Definition
```
typedef int (*CVSpilsJacTimesVecFnB)(N_Vector vB, N_Vector JvB,
                                     realtype t, N_Vector y, N_Vector yB,
                                     N_Vector fyB, void *user_dataB,
                                     N_Vector tmpB);
```

Purpose This function computes the action of the Jacobian JB for the backward problem on a given vector vB.

Arguments
- vB is the vector by which the Jacobian must be multiplied to the right.
- JvB is the computed output vector JB*vB.
- t is the current value of the independent variable.
- y is the current value of the forward solution vector.
- yB is the current value of the backward dependent variable vector.
- fyB is the current value of the backward right-hand side function f_B.
- user_dataB is a pointer to user data – the same as passed to CVodeSetUserDataB.
- tmpB is a pointer to memory allocated for a variable of type N_Vector which can be used by CVSpilsJacTimesVecFn as temporary storage or work space.

Return value The return value of a function of type CVSpilsJtimesFnB should be 0 if successful or nonzero if an error was encountered, in which case the integration is halted.

Notes A user-supplied Jacobian-vector product function must load the vector JvB with the product of the Jacobian of the backward problem at the point (t,y, yB) and the vector vB. Here, y is the solution of the original IVP at time t and yB is the solution of the backward problem at the same time. The rest of the arguments are equivalent to those passed to a function of type CVSpilsJacTimesVecFn (see §4.6.8). If the backward problem is the adjoint of $\dot{y} = f(t, y)$, then this function is to compute $-(\partial f/\partial y)^T v_B$.

CVSpilsJacTimesVecFnBS

Definition
```
typedef int (*CVSpilsJacTimesVecFnBS)(N_Vector vB, N_Vector JvB,
                                      realtype t, N_Vector y, N_Vector *yS,
                                      N_Vector yB, N_Vector fyB,
                                      void *user_dataB, N_Vector tmpB);
```

Purpose: This function computes the action of the Jacobian JB for the backward problem on a given vector vB, in the case where the backward problem depends on the forward sensitivities.

Arguments:
- vB is the vector by which the Jacobian must be multiplied to the right.
- JvB is the computed output vector JB*vB.
- t is the current value of the independent variable.
- y is the current value of the forward solution vector.
- yS is a pointer to an array containing the forward sensitivity vectors.
- yB is the current value of the backward dependent variable vector.
- fyB is the current value of the backward right-hand side function f_B.
- user_dataB is a pointer to user data – the same as passed to CVodeSetUserDataB.
- tmpB is a pointer to memory allocated for a variable of type N_Vector which can be used by CVSpilsJacTimesVecFn as temporary storage or work space.

Return value: The return value of a function of type CVSpilsJtimesFnBS should be 0 if successful or nonzero if an error was encountered, in which case the integration is halted.

Notes: A user-supplied Jacobian-vector product function must load the vector JvB with the product of the Jacobian of the backward problem at the point (t,y, yB) and the vector vB. Here, y is the solution of the original IVP at time t and yB is the solution of the backward problem at the same time. The rest of the arguments are equivalent to those passed to a function of type CVSpilsJacTimesVecFn (see §4.6.8).

6.3.9 Preconditioning for the backward problem (linear system solution)

If preconditioning is used during integration of the backward problem, then the user must provide a C function to solve the linear system $Pz = r$, where P may be either a left or a right preconditioner matrix. Here P should approximate (at least crudely) the Newton matrix $M_B = I - \gamma_B J_B$, where $J_B = \partial f_B / \partial y_B$. If preconditioning is done on both sides, the product of the two preconditioner matrices should approximate M_B. This function must be of one of the following two types:

CVSpilsPrecSolveFnB

Definition
```
typedef int (*CVSpilsPrecSolveFnB)(realtype t, N_Vector y,
                                   N_Vector yB, N_Vector fyB,
                                   N_Vector rvecB, N_Vector zvecB,
                                   realtype gammaB, realtype deltaB,
                                   void *user_dataB, N_Vector tmpB);
```

Purpose: This function solves the preconditioning system $Pz = r$ for the backward problem.

Arguments:
- t is the current value of the independent variable.
- y is the current value of the forward solution vector.
- yB is the current value of the backward dependent variable vector.

fyB	is the current value of the backward right-hand side function f_B.
rvecB	is the right-hand side vector r of the linear system to be solved.
zvecB	is the computed output vector.
gammaB	is the scalar appearing in the Newton matrix, $M_B = I - \gamma_B J_B$.
deltaB	is an input tolerance to be used if an iterative method is employed in the solution.
user_dataB	is a pointer to user data — the same as the user_dataB parameter passed to CVodeSetUserDataB.
tmpB	is a pointer to memory allocated for a variable of type N_Vector which can be used for work space.

Return value The return value of a preconditioner solve function for the backward problem should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

CVSpilsPrecSolveFnBS

Definition
```
typedef int (*CVSpilsPrecSolveFnBS)(realtype t, N_Vector y, N_Vector *yS,
                                    N_Vector yB, N_Vector fyB,
                                    N_Vector rvecB, N_Vector zvecB,
                                    realtype gammaB, realtype deltaB,
                                    void *user_dataB, N_Vector tmpB);
```

Purpose This function solves the preconditioning system $Pz = r$ for the backward problem, in the case where the backward problem depends on the forward sensitivities.

Arguments

t	is the current value of the independent variable.
y	is the current value of the forward solution vector.
yS	is a pointer to an array containing the forward sensitivity vectors.
yB	is the current value of the backward dependent variable vector.
fyB	is the current value of the backward right-hand side function f_B.
rvecB	is the right-hand side vector r of the linear system to be solved.
zvecB	is the computed output vector.
gammaB	is the scalar appearing in the Newton matrix, $M_B = I - \gamma_B J_B$.
deltaB	is an input tolerance to be used if an iterative method is employed in the solution.
user_dataB	is a pointer to user data — the same as the user_dataB parameter passed to CVodeSetUserDataB.
tmpB	is a pointer to memory allocated for a variable of type N_Vector which can be used for work space.

Return value The return value of a preconditioner solve function for the backward problem should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

6.3.10 Preconditioning for the backward problem (Jacobian data)

If the user's preconditioner requires that any Jacobian-related data be preprocessed or evaluated, then this needs to be done in a user-supplied C function of one of the following two types:

CVSpilsPrecSetupFnB

Definition
```
typedef int (*CVSpilsPrecSetupFnB)(realtype t, N_Vector y,
                                   N_Vector yB, N_Vector fyB,
                                   booleantype jokB, booleantype *jcurPtrB,
                                   realtype gammaB, void *user_dataB,
                                   N_Vector tmp1B, N_Vector tmp2B,
                                   N_Vector tmp3B);
```

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

Arguments The arguments of a CVSpilsPrecSetupFnB are as follows:

- t is the current value of the independent variable.
- y is the current value of the forward solution vector.
- yB is the current value of the backward dependent variable vector.
- fyB is the current value of the backward right-hand side function f_B.
- jokB is an input flag indicating whether Jacobian-related data needs to be recomputed (jokB=FALSE) or information saved from a previous invokation can be safely used (jokB=TRUE).
- jcurPtr is an output flag which must be set to TRUE if Jacobian-relatd data was recomputed or FALSE otherwise.
- gammaB is the scalar appearing in the Newton matrix.
- user_dataB is a pointer to user data — the same as the user_dataB parameter passed to CVodeSetUserDataB.
- tmp1B
- tmp2B
- tmp3B are pointers to memory allocated for vectors which can be used as temporary storage or work space.

Return value The return value of a preconditioner setup function for the backward problem should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

CVSpilsPrecSetupFnBS

Definition
```
typedef int (*CVSpilsPrecSetupFnBS)(realtype t, N_Vector y, N_Vector *yS,
                                    N_Vector yB, N_Vector fyB,
                                    booleantype jokB, booleantype *jcurPtrB,
                                    realtype gammaB, void *user_dataB,
                                    N_Vector tmp1B, N_Vector tmp2B,
                                    N_Vector tmp3B);
```

Purpose This function preprocesses and/or evaluates Jacobian-related data needed by the preconditioner for the backward problem, in the case where the backward problem depends on the forward sensitivities.

Arguments The arguments of a CVSpilsPrecSetupFnBS are as follows:

- t is the current value of the independent variable.
- y is the current value of the forward solution vector.
- yS is a pointer to an array containing the forward sensitivity vectors.
- yB is the current value of the backward dependent variable vector.
- fyB is the current value of the backward right-hand side function f_B.
- jokB is an input flag indicating whether Jacobian-related data needs to be recomputed (jokB=FALSE) or information saved from a previous invokation can be safely used (jokB=TRUE).

- `jcurPtr` is an output flag which must be set to `TRUE` if Jacobian-relatd data was recomputed or `FALSE` otherwise.
- `gammaB` is the scalar appearing in the Newton matrix.
- `user_dataB` is a pointer to user data — the same as the `user_dataB` parameter passed to `CVodeSetUserDataB`.
- `tmp1B`
- `tmp2B`
- `tmp3B` are pointers to memory allocated for vectors which can be used as temporary storage or work space.

Return value The return value of a preconditioner setup function for the backward problem should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

6.4 Using CVODES preconditioner modules for the backward problem

As on the forward integration phase, the efficiency of Krylov iterative methods for the solution of linear systems can be greatly enhanced through preconditioning. Both preconditioner modules provided with SUNDIALS, the serial banded preconditioner CVBANDPRE and the parallel band-block-diagonal preconditioner module CVBBDPRE, provide interface functions through which they can be used on the backward integration phase.

6.4.1 Using the banded preconditioner CVBANDPRE

The adjoint module in CVODES offers an interface to the banded preconditioner module CVBANDPRE described in section §4.8.1

CVSPILS_MEM_NULL	The cvode_mem argument was NULL.
CVSPILS_LMEM_NULL	No linear solver has been attached.
CVSPILS_ILL_INPUT	An invalid parameter has been passed.

For more details on CVBANDPRE see §4.8.1.

6.4.2 Using the band-block-diagonal preconditioner CVBBDPRE

The adjoint module in CVODES offers an interface to the band-block-diagonal preconditioner module CVBBDPRE described in section §4.8.2. This generates a preconditioner that is a block-diagonal matrix with each block being a band matrix and can be used with one of the Krylov linear solvers and with the MPI-parallel vector module NVECTOR_PARALLEL.

In order to use the CVBBDPRE module in the solution of the backward problem, the user must define one or two additional functions, described at the end of this section.

6.4.2.1 Initialization of CVBBDPRE

The CVBBDPRE module is initialized by calling the following function, *after* one of the CVSPILS linear solvers has been specified by calling the appropriate function (see §6.2.5).

`CVBBDPrecInitB`

Call
```
flag = CVBBDPrecInitB(cvode_mem, which, NlocalB, mudqB, mldqB,
                      mukeepB, mlkeepB, dqrelyB, glocB, gcommB);
```

Description The function `CVBBDPrecInitB` initializes and allocates memory for the CVBBDPRE preconditioner for the backward problem. It creates, allocates, and stores (internally in the CVODES solver block) a pointer to the newly created CVBBDPRE memory block.

Arguments

cvode_mem	(void *) pointer to the CVODES memory block.
which	(int) the identifier of the backward problem.
NlocalB	(long int) local vector dimension for the backward problem.
mudqB	(long int) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
mldqB	(long int) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
mukeepB	(long int) upper half-bandwidth of the retained banded approximate Jacobian block.
mlkeepB	(long int) lower half-bandwidth of the retained banded approximate Jacobian block.
dqrelyB	(realtype) the relative increment in components of yB used in the difference quotient approximations. The default is dqrelyB= $\sqrt{\text{unit roundoff}}$, which can be specified by passing dqrely= 0.0.
glocB	(CVBBDLocalFnB) the C function which computes the function $g_B(t, y, y_B)$ approximating the right-hand side of the backward problem.
gcommB	(CVBBDCommFnB) the optional C function which performs all interprocess communication required for the computation of g_B.

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

CVSPILS_SUCCESS	The call to CVodeBBDPrecInitB was successful.
CVSPILS_MEM_FAIL	A memory allocation request has failed.
CVSPILS_MEM_NULL	The cvode_mem argument was NULL.
CVSPILS_LMEM_NULL	No linear solver has been attached.
CVSPILS_ILL_INPUT	An invalid parameter has been passed.

To reinitialize the CVBBDPRE preconditioner module for the backward problem, possibly with changes in `mudqB`, `mldqB`, or `dqrelyB`, call the following function:

`CVBBDPrecReInitB`

Call `flag = CVBBDPrecReInitB(cvode_mem, which, mudqB, mldqB, dqrelyB);`

Description The function `CVBBDPrecReInitB` reinitializes the CVBBDPRE preconditioner for the backward problem.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block returned by `CVodeCreate`.
- `which` (`int`) the identifier of the backward problem.
- `mudqB` (`long int`) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
- `mldqB` (`long int`) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
- `dqrelyB` (`realtype`) the relative increment in components of `yB` used in the difference quotient approximations.

Return value The return value `flag` (of type `int`) is one of:
- `CVSPILS_SUCCESS` The call to `CVodeBBDPrecReInitB` was successful.
- `CVSPILS_MEM_FAIL` A memory allocation request has failed.
- `CVSPILS_MEM_NULL` The `cvode_mem` argument was `NULL`.
- `CVSPILS_PMEM_NULL` The `CVodeBBDPrecInitB` has not been previously called.
- `CVSPILS_LMEM_NULL` No linear solver has been attached.
- `CVSPILS_ILL_INPUT` An invalid parameter has been passed.

For more details on CVBBDPRE see §4.8.2.

6.4.2.2 User-supplied functions for CVBBDPRE

To use the CVBBDPRE module, the user must supply one or two functions which the module calls to construct the preconditioner: a required function `glocB` (of type `CVBBDLocalFnB`) which approximates the right-hand side of the backward problem and which is computed locally, and an optional function `gcommB` (of type `CVBBDCommFnB`) which performs all interprocess communication necessary to evaluate this approximate right-hand side (see §4.8.2). The prototypes for these two functions are described below.

`CVBBDLocalFnB`

Definition
```
typedef int (*CVBBDLocalFnB)(long int NlocalB, realtype t, N_Vector y,
                             N_Vector yB, N_Vector gB, void *user_dataB);
```

Purpose This `glocB` function loads the vector `gB`, an approximation to the right-hand side f_B of the backward problem, as a function of `t`, `y`, and `yB`.

Arguments
- `NlocalB` is the local vector length for the backward problem.
- `t` is the value of the independent variable.
- `y` is the current value of the forward solution vector.
- `yB` is the current value of the backward dependent variable vector.
- `gB` is the output vector, $g_B(t, y, y_B)$.
- `user_dataB` is a pointer to user data — the same as the `user_dataB` parameter passed to `CVodeSetUserDataB`.

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

Notes This routine must assume that all interprocess communication of data needed to calculate `gB` has already been done, and this data is accessible within `user_dataB`.

Before calling the user's `CVBBDLocalFnB`, CVODES needs to evaluate (through interpolation) the values of the states from the forward integration. If an error occurs in the interpolation, CVODES triggers an unrecoverable failure in the preconditioner setup function which will halt the integration (`CVodeB` returns `CV_LSETUP_FAIL`).

`CVBBDCommFnB`

Definition
```
typedef int (*CVBBDCommFnB)(long int NlocalB, realtype t, N_Vector y,
                            N_Vector yB, void *user_dataB);
```

Purpose This `gcommB` function must perform all interprocess communications necessary for the execution of the `glocB` function above, using the input vectors `y` and `yB`.

Arguments
- `NlocalB` is the local vector length.
- `t` is the value of the independent variable.
- `y` is the current value of the forward solution vector.
- `yB` is the current value of the backward dependent variable vector.
- `user_dataB` is a pointer to user data — the same as the `user_dataB` parameter passed to `CVodeSetUserDataB`.

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

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

Each call to the `gcommB` function is preceded by a call to the function that evaluates the right-hand side of the backward problem with the same `t`, `y`, and `yB`, arguments. If there is no additional communication needed, then pass `gcommB = NULL` to `CVBBDPrecInitB`.

Chapter 7

Description of the NVECTOR module

The SUNDIALS solvers are written in a data-independent manner. They all operate on generic vectors (of type N_Vector) through a set of operations defined by the particular NVECTOR implementation. Users can provide their own specific implementation of the NVECTOR module, or use one of 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    (*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 7.1 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_VCloneEmptyVectorArray`. 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_VCloneEmptyVectorArray(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`.

Table 7.1: Description of the NVECTOR operations

Name	Usage and Description
`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.
`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`: $z_i = c$, $i = 0, \ldots, n-1$.

continued on next page

continued from last page

Name	Usage and Description
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`: $y_i = \|x_i\|$, $i = 0, \ldots, n-1$.
N_VInv	`N_VInv(x, z);` Sets the components of the `N_Vector` `z` to be the inverses of the components of the `N_Vector` `x`: $z_i = 1.0/x_i$, $i = 0, \ldots, n-1$. 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`: $m = \max_i \|x_i\|$.
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`: $m = \min_i x_i$.

continued on next page

continued from last page	
Name	**Usage and Description**
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.
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.

7.1 The NVECTOR_SERIAL implementation

The serial implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_SERIAL, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, and a boolean flag *own_data* which specifies the ownership of *data*.

```
struct _N_VectorContent_Serial {
  long int length;
  booleantype own_data;
  realtype *data;
};
```

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 7.1. Their names are obtained from those in Table 7.1 by appending the suffix `_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`

- `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_VCloneEmptyVectorArray_Serial`.

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

- `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_VCloneEmptyVectorArray_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.

7.2 The NVECTOR_PARALLEL implementation

The NVECTOR_PARALLEL implementation of the NVECTOR module provided with SUNDIALS is based on MPI. It defines the *content* field of N

viitihe

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

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

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

Implementation:

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

- `NV_COMM_P`

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

  Implementation:

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

- `NV_Ith_P`

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

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

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

  Implementation:

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

The NVECTOR_PARALLEL module defines parallel implementations of all vector operations listed in Table 7.1 Their names are obtained from those in Table 7.1 by appending the suffix `_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.

```
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_VCloneEmptyVectorArray_Parallel`

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

  ```
  N_Vector *N_VCloneEmptyVectorArray_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_VCloneEmptyVectorArray_Parallel`.

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

- `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_VCloneEmptyVectorArray_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.

7.3 The NVECTOR_OPENMP implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR_OPENMP, and an implementation using Pthreads, called NVECTOR_PTHREADS. Testing has shown that vectors should be of length at least $100,000$ before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The OpenMP NVECTOR implementation provided with SUNDIALS, NVECTOR_OPENMP, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag *own_data* which specifies the ownership of *data*, and the number of threads. Operations on the vector are threaded using OpenMP.

```
struct _N_VectorContent_OpenMP {
  long int length;
  booleantype own_data;
  realtype *data;
  int num_threads;
};
```

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_

- `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.

  ```
  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_VCloneEmptyVectorArray_OpenMP`

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

  ```
  N_Vector *N_VCloneEmptyVectorArray_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_VCloneEmptyVectorArray_OpenMP`.

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

- `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_VCloneEmptyVectorArray_OpenMP` set the field

```
struct _N_VectorContent_Pthreads {
  long int length;
  booleantype own_data;
  realtype *data;
  int num_threads;
};
```

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 7.1. Their names are obtained from those in Table 7.1 by appending the suffix _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.

  `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_VCloneEmptyVectorArray_Pthreads`

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

  `N_Vector *N_VCloneEmptyVectorArray_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_VCloneEmptyVectorArray_Pthreads`.

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

- `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_VCloneEmptyVectorArray_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. !

7.5 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.

7.6 NVECTOR functions used by CVODES

In Table 7.2 below, we list the vector functions in the NVECTOR module used within the CVODES package. The table also shows, for each function, which of the code modules uses the function. The CVODES column shows function usage within the main integrator module, while the remaining sevem columns show function usage within each of the eight CVODES linear solvers, the CVBANDPRE and CVBBDPRE preconditioner modules, and the CVODES adjoint sensitivity module (denoted here by CVODEA). 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 `cvodes_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 `cvodes_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 CVODES 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 CVODES user does not need to know anything about the usage of vector functions by the CVODES code modules in order to use CVODES. The information is presented as an implementation detail for the interested reader.

The vector functions listed in Table 7.1 that are *not* used by CVODES are: `N_VWL2Norm`, `N_VL1Norm`, `N_VWrmsNormMask`, `N_VConstrMask`, `N_VCloneEmpty`, and `N_VMinQuotient`. Therefore a user-supplied NVECTOR module for CVODES could omit these six kernels.

Table 7.2: List of vector functions usage by CVODES code modules

	CVODES	CVDLS	CVDIAG	CVSPILS	CVSLS	CVBANDPRE	CVBBDPRE	CVODEA
N_VClone	✓		✓	✓				✓
N_VDestroy	✓		✓	✓				✓
N_VCloneVectorArray	✓							✓
N_VDestroyVectorArray	✓							✓
N_VSpace	✓							
N_VGetArrayPointer		✓			✓	✓	✓	
N_VSetArrayPointer		✓						
N_VLinearSum	✓	✓	✓	✓				✓
N_VConst	✓			✓				
N_VProd	✓		✓	✓				
N_VDiv	✓		✓	✓				
N_VScale	✓	✓	✓	✓	✓	✓	✓	✓
N_VAbs	✓							
N_VInv	✓		✓					
N_VAddConst	✓		✓					
N_VDotProd				✓				
N_VMaxNorm	✓							
N_VWrmsNorm	✓	✓		✓		✓	✓	
N_VMin	✓							
N_VCompare			✓					
N_VInvTest			✓					

Chapter 8

Providing Alternate Linear Solver Modules

The central CVODES module interfaces a the linear solver module by way of calls to four functions. These are denoted here by `linit`, `lsetup`, `lsolve`, and `lfree` block, of type `CVodeMem`, is not directly accessible to the specification function, but rather is itself a field in the CVODES memory block. For a given backward problem identifier `which`, the corresponding memory block must be located in the linked list starting at `cvode_mem->cv_adj_mem->cvB_mem`; see for example the function `CVDenseB` for specific details. This specification function must also allocate the linear solver memory for the backward problem, and attach that, as well as a corresponding memory free function, to the above block `cvB_mem`, of type `struct CVodeBMemRec`. The specification function for backward integration should return a negative value if it encounters an illegal input, if backward integration has not been initialized, or if its memory allocation failed.

These four functions, which interface between CVODES and the linear solver module, necessarily have fixed call sequences. Thus, a user wishing to implement another linear solver within the CVODES 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 CVODES memory block, by which the function can access various data related to the CVODES solution. The contents of this memory block are given in the file `cvodes_impl.h` (but not reproduced here, for the sake of space).

8.1 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 CVODES memory pointer of type `CVodeMem`.

Return value An `linit` function should return 0 if it has successfully initialized the CVODES linear solver, and a negative value otherwise.

8.2 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.(2.6)). 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 $\partial f/\partial y$. Alterntively, 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 CVODES 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 CVODES 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 CVODES 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 CVODES 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.

8.3 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.(2.6)), 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 CVODES 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.(2.7). 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, $f(t_n, \texttt{ycur})$.

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

Chapter 9

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 paage, a lnrmhpdvastsededrsedlvputnoatrix7.407111a 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.

9.1 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.

9.1.1 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 9.1 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`.

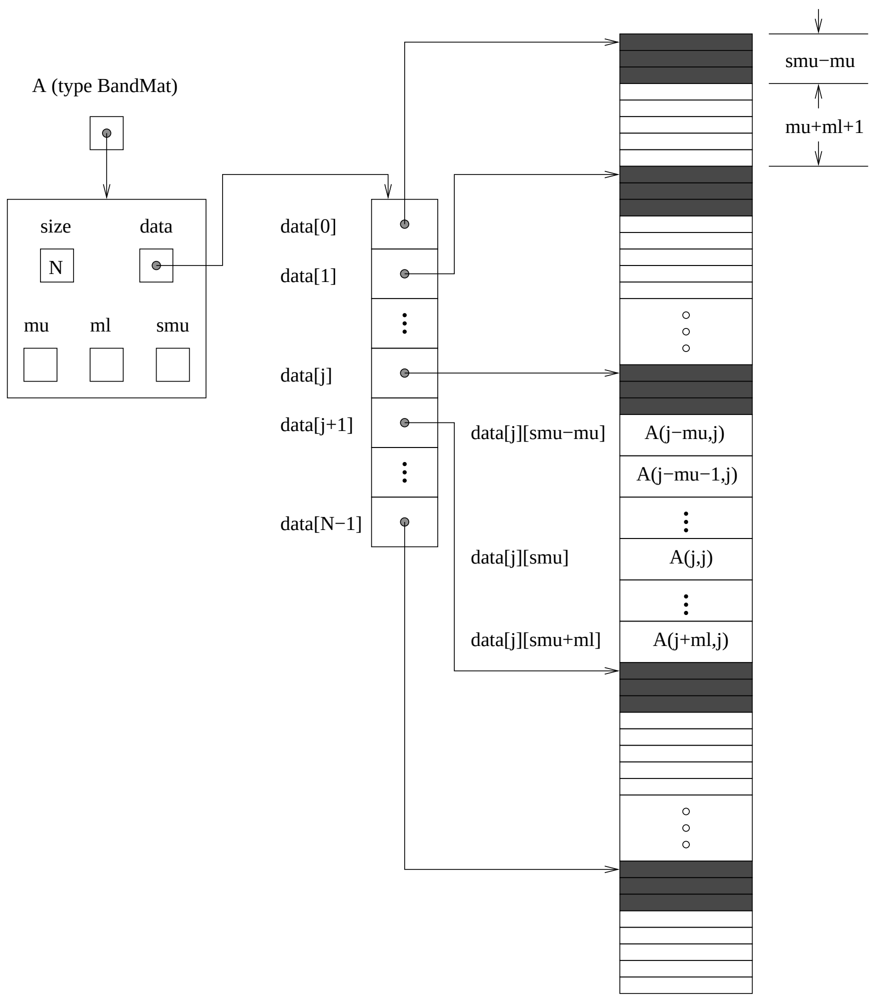


Figure 9.1: Diagram of the storage for a banded matrix of type `DlsMat`. Here `A` is an $N \times N$ band matrix of type `DlsMat` with upper and lower half-bandwidths `mu` and `ml`, respectively. The rows and columns of `A` are numbered from 0 to $N-1$ and the (i, j)-th element of `A` is denoted `A(i,j)`. The greyed out areas of the underlying component storage are used by the `BandGBTRF` and `BandGBTRS` routines.

9.1.2 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`, $0 \leq$ `i` $< M$, $0 \leq$ `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`, $0 \leq$ `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 $0 \leq$ `i`, `j` $\leq N - 1$. The location (`i`,`j`) should further satisfy `j`$-$`(A->mu)` $\leq$ `i` $\leq$ `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`, $0 \leq$ `j` $\leq N - 1$. 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)` $\leq$ `i` $\leq$ `j`$+$`(A->ml)`.

9.1.3 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.
- `NewLintArray`: allocation of an array of `long int` 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`;

- `newLintArray`

  `newLintArray(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 `newLintArray`, `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`.

9.1.4 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.
- `NewLintArray`: allocation of an array of `int` 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`;

- `newLintArray`

  `newLintArray(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 `newLintArray`, `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`.

9.2 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`.

9.2.1 Type SlsMat

The type `SlsMat`, defined in `sundials_sparse.h` is a pointer to a structure defining a generic compressed-sparse-column matrix, and is used with all linear solvers in the *sls* family:

```
typedef struct _SlsMat {
  int M;
  int N;
  int NNZ;
  realtype *data;
  int *rowvals;
  int *colptrs;
} *SlsMat;
```

The fields of this structure are as follows (see Figure 9.2 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)

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

rowvals - pointer to a contiguous block of `int` variables (of length `NNZ`), containing the row indices of each nonzero entry held in `data`

colptrs - pointer to a contiguous block of `int` variables (of length `N+1`). Each entry provides the index of the first column entry into the `data` and `rowvals` arrays, e.g. if `colptr[3]=7`, then the first nonzero entry in the fourth column of the matrix is located in `data[7]`, and is located in row `rowvals[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 `rowvals` arrays.

For example, the 5×4 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;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0};
rowvals = {1, 3, 0, 2, 0, 1, 3, 4};
colptrs = {0, 2, 4, 5, 8};
```

or

```
M = 5;
N = 4;
NNZ = 10;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0, *, *};
rowvals = {1, 3, 0, 2, 0, 1, 3, 4, *, *};
colptrs = {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 `colptrs` 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.

9.2.2 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`.

- `NewSparseMat`

  `NewSparseMat(M, N, NNZ)` allocates storage for an `M` by `N` sparse matrix, with storage for up to `NNZ` nonzero entries.

- `SlsConvertDls`

  `SlsConvertDls(A)` converts a dense or band matrix `A` of type `DlsMat` into a new sparse matrix of type `SlsMat` by retaining only the nonzero values of the matrix `A`.

- `DestroySparseMat`

  `DestroySparseMat(A)` frees the memory for a sparse matrix `A` allocated by either `NewSparseMat` or `SlsConvertDls`.

- `SlsSetToZero(A)` zeros out the `SlsMat` matrix `A`. The storage for `A` is left unchanged.

- `CopySparseMat`

  `CopySparseMat(A, B)` copies the `SlsMat A` into the `SlsMat B`. It is assumed that the matrices have the same row/column dimensions. If `B` has insufficient storage to hold all the nonzero entries of `A`, the data and row index arrays in `B` are reallocated to match those in `A`.

- `ScaleSparseMat`

  `ScaleSparseMat(c, A)` scales every element in the `SlsMat A` by the `realtype` scalar `c`.

- `AddIdentitySparseMat`

  `AddIdentitySparseMat(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.

- `SlsAddMat`

  `SlsAddMat(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.

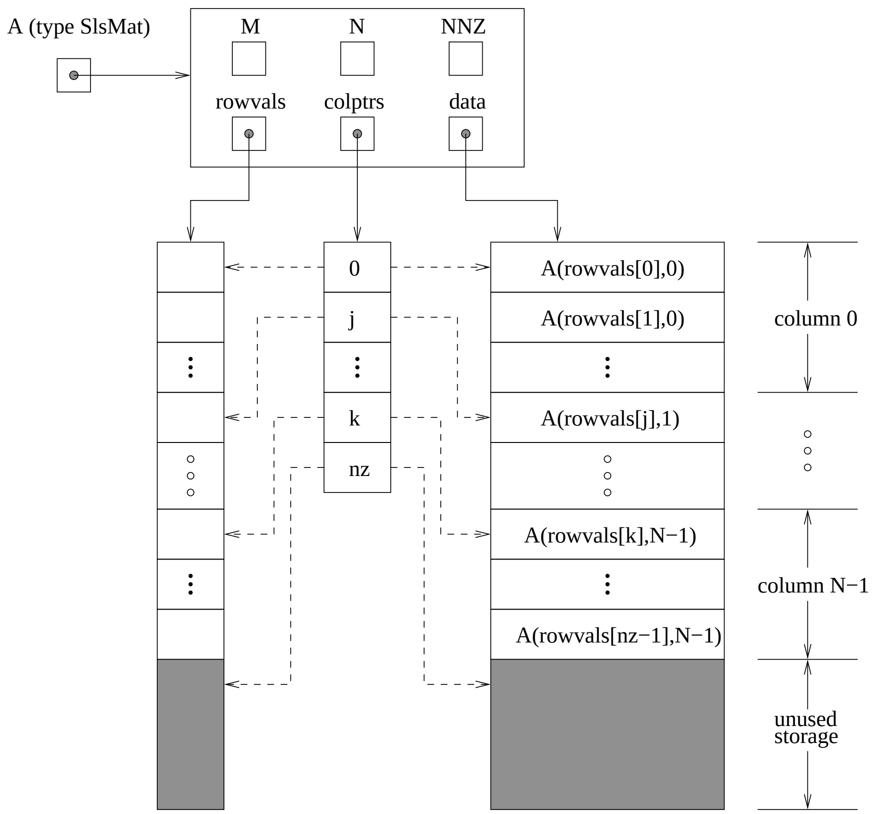


Figure 9.2: Diagram of the storage for a compressed-sparse-column matrix of type `SlsMat`. Here `A` is an $M \times N$ sparse matrix of type `SlsMat` with storage for up to `NNZ` nonzero entries (the allocated length of both `data` and `rowvals`). The entries in `rowvals` may assume values from 0 to $M-1$, 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 `colptrs` array contains $N+1$ entries; the first N denote the starting index of each column within the `rowvals` 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 `rowvals` indicate extra allocated space.

- `ReallocSparseMat`

  `ReallocSparseMat(A)` eliminates unused storage in the `SlsMat A` by resizing the internal `data` and `rowvals` arrays to contain exactly `colptrs[N]` values.

- `SlsMatvec`

  `SlsMatvec(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.

- `PrintSparseMat`

  `PrintSparseMat(A)` Prints the `SlsMat` matrix `A` to standard output.

9.2.3 The KLU solver

KLU is a sparse matrix factorization and solver library written by Tim Davis [1, 12]. 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 7.1, 7.3 and 7.4 for details).

9.2.4 The SUPERLUMT solver

SUPERLUMT is a threaded sparse matrix factorization and solver library written by X. Sherry Li [2, 26, 13]. 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 7.1, 7.3 and 7.4 for details).

9.3 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.

9.3.1 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`.

9.3.2 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`.

9.3.3 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`.

9.3.4 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.

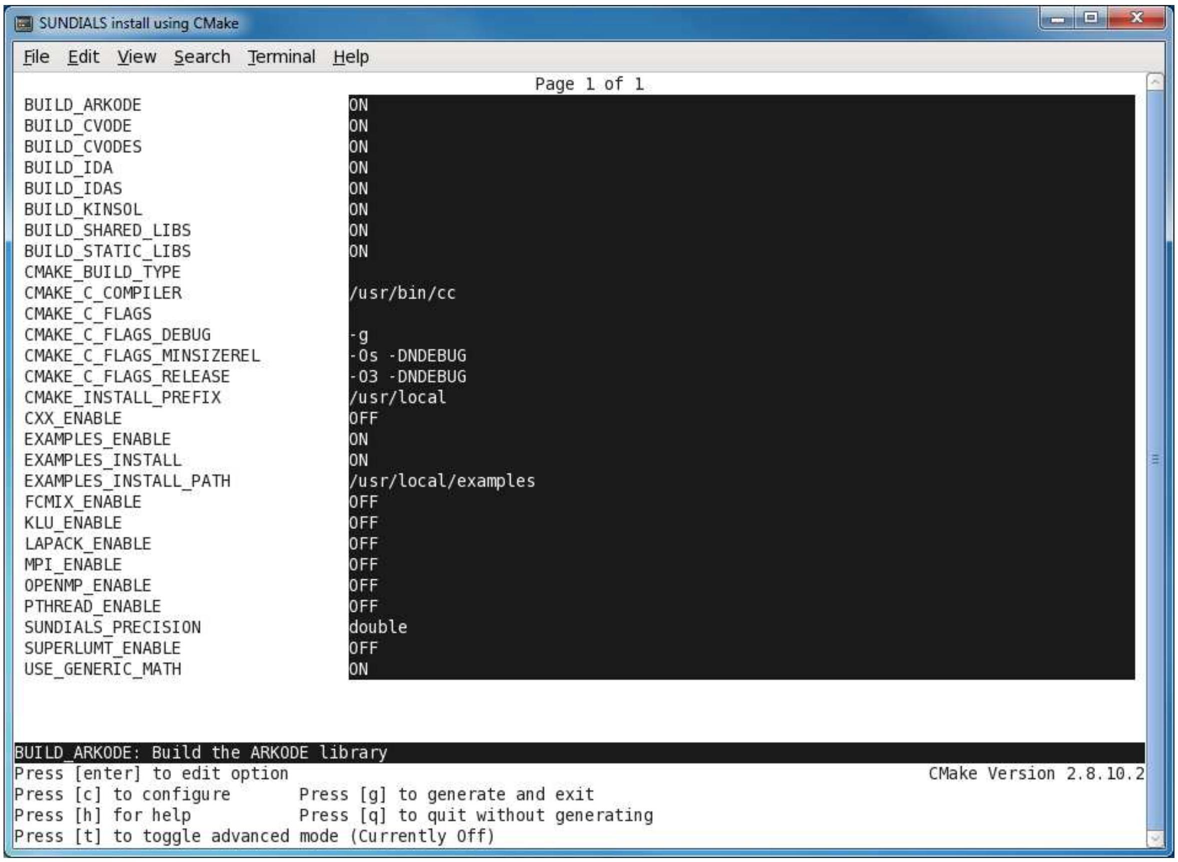


Figure A.1: Default configuration screen. Note: Initial screen is empty. To get this default configuration, press 'c' repeatedly (accepting default values denoted with asterisk) until the 'g' option is available.

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
```

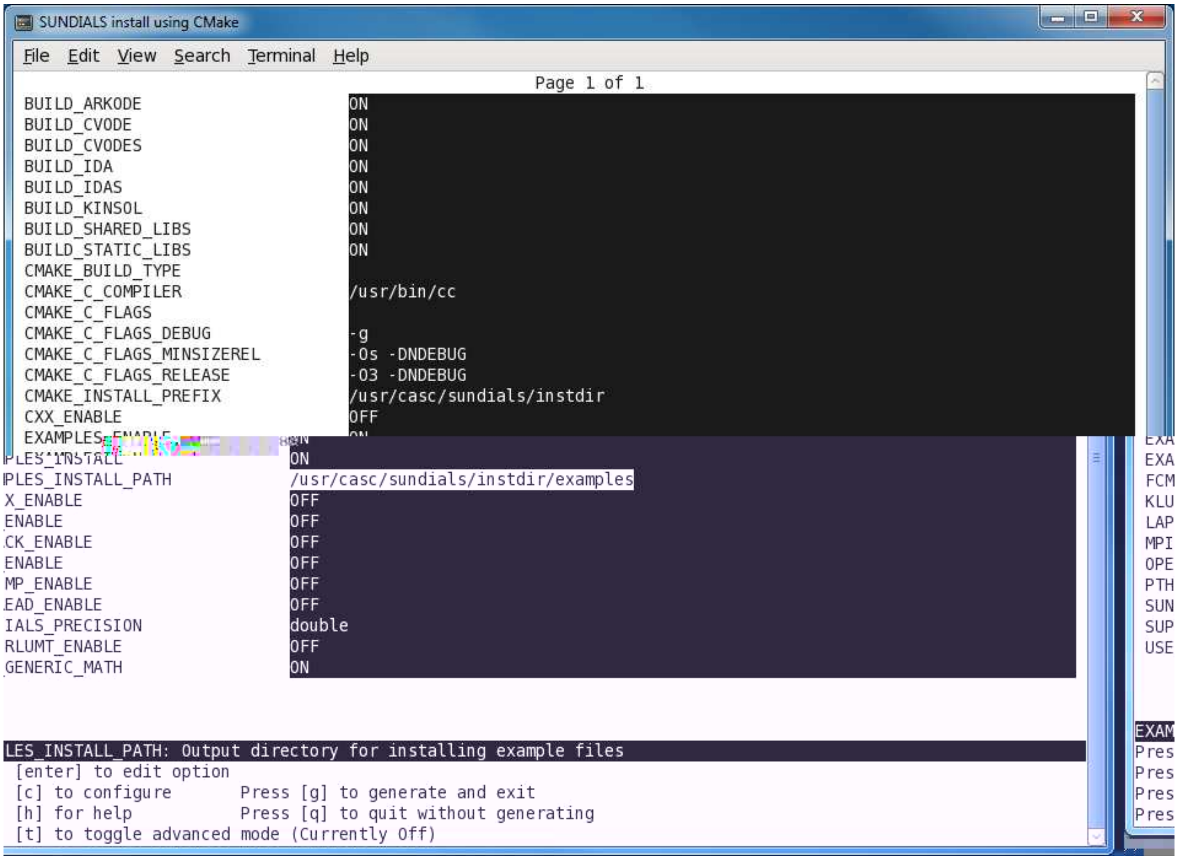


Figure A.2: Changing the *instdir* for SUNDIALS and corresponding examples

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

`KLU_ENABLE` - Enable KLU support
Default: OFF

`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.

`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

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 4.2.1. 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 2.4. 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.

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	

Appendix B

CVODES Constants

Below we list all input and output constants used by the main solver and linear solver modules, together with their numerical values and a short description of their meaning.

B.1 CVODES input constants

CVODES main solver module

CV_ADAMS	1	Adams-Moulton linear multistep method.
CV_BDF	2	BDF linear multistep method.
CV_FUNCTIONAL	1	Nonlinear system solution through functional iterations.
CV_NEWTON	2	Nonlinear system solution through Newton iterations.
CV_NORMAL	1	Solver returns at specified output time.
CV_ONE_STEP	2	Solver returns after each successful step.
CV_SIMULTANEOUS	1	Simultaneous corrector forward sensitivity method.
CV_STAGGERED	2	Staggered corrector forward sensitivity method.
CV_STAGGERED1	3	Staggered (variant) corrector forward sensitivity method.
CV_CENTERED	1	Central difference quotient approximation (2^{nd} order) of the sensitivity RHS.
CV_FORWARD	2	Forward difference quotient approximation (1^{st} order) of the sensitivity RHS.

CVODES adjoint solver module

CV_HERMITE	1	Use Hermite interpolation.
CV_POLYNOMIAL	2	Use variable-degree polynomial interpolation.

Iterative linear solver module

PREC_NONE	0	No preconditioning
PREC_LEFT	1	Preconditioning on the left only.
PREC_RIGHT	2	Preconditioning on the right only.
PREC_BOTH	3	Preconditioning on both the left and the right.
MODIFIED_GS	1	Use modified Gram-Schmidt procedure.
CLASSICAL_GS	2	Use classical Gram-Schmidt procedure.

B.2 CVODES output constants

CVODES **main solver module**		
CV_SUCCESS	0	Successful function return.
CV_TSTOP_RETURN	1	`CVode` succeeded by reaching the specified stopping point.
CV_ROOT_RETURN	2	`CVode` succeeded and found one or more roots.
CV_WARNING	99	`CVode` succeeded but an unusual situation occurred.
CV_TOO_MUCH_WORK	-1	The solver took `mxstep` internal steps but could not reach tout.
CV_TOO_MUCH_ACC	-2	The solver could not satisfy the accuracy demanded by the user for some internal step.
CV_ERR_FAILURE	-3	Error test failures occurred too many times during one internal time step or minimum step size was reached.
CV_CONV_FAILURE	-4	Convergence test failures occurred too many times during one internal time step or minimum step size was reached.
CV_LINIT_FAIL	-5	The linear solver's initialization function failed.
CV_LSETUP_FAIL	-6	The linear solver's setup function failed in an unrecoverable manner.
CV_LSOLVE_FAIL	-7	The linear solver's solve function failed in an unrecoverable manner.
CV_RHSFUNC_FAIL	-8	The right-hand side function failed in an unrecoverable manner.
CV_FIRST_RHSFUNC_ERR	-9	The right-hand side function failed at the first call.
CV_REPTD_RHSFUNC_ERR	-10	The right-hand side function had repetead recoverable errors.
CV_UNREC_RHSFUNC_ERR	-11	The right-hand side function had a recoverable error, but no recovery is possible.
CV_RTFUNC_FAIL	-12	The rootfinding function failed in an unrecoverable manner.
CV_MEM_FAIL	-20	A memory allocation failed.
CV_MEM_NULL	-21	The `cvode_mem` argument was `NULL`.
CV_ILL_INPUT	-22	One of the function inputs is illegal.
CV_NO_MALLOC	-23	The CVODE memory block was not allocated by a call to `CVodeMalloc`.
CV_BAD_K	-24	The derivative order k is larger than the order used.
CV_BAD_T	-25	The time t is outside the last step taken.
CV_BAD_DKY	-26	The output derivative vector is `NULL`.
CV_TOO_CLOSE	-27	The output and initial times are too close to each other.
CV_NO_QUAD	-30	Quadrature integration was not activated.
CV_QRHSFUNC_FAIL	-31	The quadrature right-hand side function failed in an unrecoverable manner.
CV_FIRST_QRHSFUNC_ERR	-32	The quadrature right-hand side function failed at the first call.
CV_REPTD_QRHSFUNC_ERR	-33	The quadrature ight-hand side function had repetead recoverable errors.
CV_UNREC_QRHSFUNC_ERR	-34	The quadrature right-hand side function had a recoverable error, but no recovery is possible.
CV_NO_SENS	-40	Forward sensitivity integration was not activated.
CV_SRHSFUNC_FAIL	-41	The sensitivity right-hand side function failed in an unrecoverable manner.

CV_FIRST_SRHSFUNC_ERR	-42	The sensitivity right-hand side function failed at the first call.
CV_REPTD_SRHSFUNC_ERR	-43	The sensitivity ight-hand side function had repetead recoverable errors.
CV_UNREC_SRHSFUNC_ERR	-44	The sensitivity right-hand side function had a recoverable error, but no recovery is possible.
CV_BAD_IS	-45	The sensitivity index is larger than the number of sensitivities computed.
CV_NO_QUADSENS	-50	Forward sensitivity integration was not activated.
CV_QSRHSFUNC_FAIL	-51	The sensitivity right-hand side function failed in an unrecoverable manner.
CV_FIRST_QSRHSFUNC_ERR	-52	The sensitivity right-hand side function failed at the first call.
CV_REPTD_QSRHSFUNC_ERR	-53	The sensitivity ight-hand side function had repetead recoverable errors.
CV_UNREC_QSRHSFUNC_ERR	-54	The sensitivity right-hand side function had a recoverable error, but no recovery is possible.
		CVODES adjoint solver module
CV_NO_ADJ	-101	Adjoint module was not initialized.
CV_NO_FWD	-102	The forward integration was not yet performed.
CV_NO_BCK	-103	No backward problem was specified.
CV_BAD_TB0	-104	The final time for the adjoint problem is outside the interval over which the forward problem was solved.
CV_REIFWD_FAIL	-105	Reinitialization of the forward problem failed at the first checkpoint.
CV_FWD_FAIL	-106	An error occurred during the integration of the forward problem.
CV_GETY_BADT	-107	Wrong time in interpolation function.
		CVDLS linear solver modules
CVDLS_SUCCESS	0	Successful function return.
CVDLS_MEM_NULL	-1	The cvode_mem argument was NULL.
CVDLS_LMEM_NULL	-2	The CVDLS linear solver has not been initialized.
CVDLS_ILL_INPUT	-3	The CVDLS solver is not compatible with the current NVECTOR module.
CVDLS_MEM_FAIL	-4	A memory allocation request failed.
CVDLS_JACFUNC_UNRECVR	-5	The Jacobian function failed in an unrecoverable manner.
CVDLS_JACFUNC_RECVR	-6	The Jacobian function had a recoverable error.
CVDLS_NO_ADJ	-101	The combined forward-backward problem has not been initialized.
CVDLS_LMEMB_NULL	-102	The linear solver was not initialized for the backward phase.
		CVDIAG linear solver module
CVDIAG_SUCCESS	0	Successful function return.
CVDIAG_MEM_NULL	-1	The cvode_mem argument was NULL.

CVDIAG_LMEM_NULL	-2	The CVDIAG linear solver has not been initialized.
CVDIAG_ILL_INPUT	-3	The CVDIAG solver is not compatible with the current NVECTOR module.
CVDIAG_MEM_FAIL	-4	A memory allocation request failed.
CVDIAG_INV_FAIL	-5	A diagonal element of the Jacobian was 0.
CVDIAG_RHSFUNC_UNRECVR	-6	The right-hand side function failed in an unrecoverable manner.
CVDIAG_RHSFUNC_RECVR	-7	The right-hand side function had a recoverable error.
CVDIAG_NO_ADJ	-101	The combined forward-backward problem has not been initialized.

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.
CVSLS_NO_ADJ	-101	The combined forward-backward problem has not been initialized.
CVSLS_LMEMB_NULL	-102	The linear solver was not initialized for the backward phase.

CVSPILS linear solver modules

CVSPILS_SUCCESS	0	Successful function return.
CVSPILS_MEM_NULL	-1	The `cvode_mem` argument was `NULL`.
CVSPILS_LMEM_NULL	-2	The CVSPILS linear solver has not been initialized.
CVSPILS_ILL_INPUT	-3	The CVSPILS solver is not compatible with the current NVECTOR module, or an input value was illegal.
CVSPILS_MEM_FAIL	-4	A memory allocation request failed.
CVSPILS_PMEM_NULL	-5	The preconditioner module has not been initialized.
CVSPILS_NO_ADJ	-101	The combined forward-backward problem has not been initialized.
CVSPILS_LMEMB_NULL	-102	The linear solver was not initialized for the backward phase.

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_		

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. PVODE, An ODE Solver for Parallel Computers. *Intl. J. High Perf. Comput. Apps.*, 13(4):254–365, 1999.

[8] Y. Cao, S. Li, L. R. Petzold, and R. Serban. Adjoint Sensitivity Analysis for Differential-Algebraic Equations: The Adjoint DAE System and its Numerical Solution. *SIAM J. Sci. Comput.*, 24(3):1076–1089, 2003.

[9] M. Caracotsios and W. E. Stewart. Sensitivity Analysis of Initial Value Problems with Mixed ODEs and Algebraic Equations. *Computers and Chemical Engineering*, 9:359–365, 1985.

[10] S. D. Cohen and A. C. Hindmarsh. CVODE User Guide. Technical Report UCRL-MA-118618, LLNL, September 1994.

[11] S. D. Cohen and A. C. Hindmarsh. CVODE, a Stiff/Nonstiff ODE Solver in C. *Computers in Physics*, 10(2):138–143, 1996.

[12] 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.

[13] 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.

[14] W. F. Feehery, J. E. Tolsma, and P. I. Barton. Efficient Sensitivity Analysis of Large-Scale Differential-Algebraic Systems. *Applied Numer. Math.*, 25(1):41–54, 1997.

[15] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14:470–482, 1993.

[16] K. L. Hiebert and L. F. Shampine. Implicitly Defined Output Points for Solutions of ODEs. Technical Report SAND80-0180, Sandia National Laboratories, February 1980.

[17] A. C. Hindmarsh. Detecting Stability Barriers in BDF Solvers. In J.R. Cash and I. Gladwell, editor, *Computational Ordinary Differential Equations*, pages 87–96, Oxford, 1992. Oxford University Press.

[18] A. C. Hindmarsh. Avoiding BDF Stability Barriers in the MOL Solution of Advection-Dominated Problems. *Appl. Num. Math.*, 17:311–318, 1995.

[19] A. C. Hindmarsh. The PVODE and IDA Algorithms. Technical Report UCRL-ID-141558, LLNL, December 2000.

[20] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS, suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, (31):363–396, 2005.

[21] A. C. Hindmarsh and R. Serban. Example Programs for CVODE v2.7.0. Technical report, LLNL, 2011. UCRL-SM-208110.

[22] A. C. Hindmarsh and R. Serban. User Documentation for CVODE v2.7.0. Technical Report UCRL-SM-208108, LLNL, 2011.

[23] 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.

[24] 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.

[25] S. Li, L. R. Petzold, and W. Zhu. Sensitivity Analysis of Differential-Algebraic Equations: A Comparison of Methods on a Special Problem. *Applied Num. Math.*, 32:161–174, 2000.

[26] X. S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.

[27] T. Maly and L. R. Petzold. Numerical Methods and Software for Sensitivity Analysis of Differential-Algebraic Systems. *Applied Numerical Mathematics*, 20:57–79, 1997.

[28] D.B. Ozyurt and P.I. Barton. Cheap second order directional derivatives of stiff ODE embedded functionals. *SIAM J. of Sci. Comp.*, 26(5):1725–1743, 2005.

[29] 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.

[30] 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.

[31] R. Serban and A. C. Hindmarsh. CVODES, the sensitivity-enabled ODE solver in SUNDIALS. In *Proceedings of the 5th International Conference on Multibody Systems, Nonlinear Dynamics and Control*, Long Beach, CA, 2005. ASME.

[32] R. Serban and A. C. Hindmarsh. Example Programs for CVODES v2.7.0. Technical Report UCRL-SM-208115, LLNL, 2011.

[33] 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