User Documentation for CVODES v2.2.0

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

April 2005

UCRL-SM-208111

This research was supported under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

CVODES is part of a software family called SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic equation Solvers. This suite consists of CVODE, 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 [1] and VODPK [3]. 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 [23]. VODPK is a variant of VODE that uses a preconditioned Krylov (iterative) method 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 [2]. The capabilities of both VODE and VODPK were combined in the C-language package CVODE [8, 9].

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 [5], 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 new 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.

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 and still is 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 v2.2.0

The user interface has been further refined. Several functions used for setting optional inputs were combined into a single one. 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 v2.1.0

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 §5.5.4 and §5.5.6.

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.

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

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.

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 §3.1, then §5 through §5.5.3 only, and looking at examples in [17]. In addition, to solve a forward sensitivity problem the user should read §3.2, followed by §6 through §6.2.2 only, and look at examples in [17].

In a different direction, a more advanced user with an IVP problem may want to (a) use a package preconditioner (§5.9), (b) supply his/her own Jacobian or preconditioner routines (§5.6),

(c) do multiple runs of problems of the same size (§5.5.7), (d) supply a new NVECTOR module (§8), or even (e) supply a different linear solver module (§4.2). An advanced user with a forward sensitivity problem may also want to (a) provide his/her own sensitivity equations right-hand side routine (§6.3), (b) perform multiple runs with the same number of sensitivity parameters (§6.2.1), or (c) extract additional diagnostic information (§6.2.2). A user with an adjoint sensitivity problem needs to understand the IVP solution approach at the desired level and also go through §3.3 for a short mathematical description of the adjoint approach, §7 for the usage of the adjoint module in CVODES, and the examples in [17].

The structure of this document is as follows:

- In Chapter 2 we begin with instructions for the installation of CVODES, within the structure of SUNDIALS.
- In Chapter 3, we give short descriptions of the numerical methods implemented by CVODES for the solution of initial value problems for systems of ODEs, continue with an overview of the mathematical aspects of sensitivity analysis, both forward (§3.2) and adjoint (§3.3), and conclude with a description of stability limit detection (§3.4).
- The following chapter describes the structure of the SUNDIALS suite of solvers (§4.1) and the software organization of the CVODES solver (§4.2).
- In Chapter 5, we give an overview of the usage of CVODES, as well as a complete description of the user interface and of the user-defined routines for integration of IVP ODEs. Readers that are not interested in using CVODES for sensitivity analysis can then skip to the example programs in [16].
- Chapter 6 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 5. 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 7 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 8 gives a brief overview of the generic NVECTOR module shared amongst the various components of SUNDIALS, as well as details on the two NVECTOR implementations provided with SUNDIALS: a serial implementation (§8.1) and a parallel implementation based on MPI (§8.2).
- Chapter 9 describes the specifications of linear solver modules as supplied by the user.
- Chapter 10 describes in detail the generic linear solvers shared by all SUNDIALS solvers.
- Finally, Chapter 11 lists the constants used for input to and output from CVODES.

Finally, the reader should be aware of the following notational conventions in this user guide: Program listings and identifiers (such as `CVodeMalloc`) 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. In the Index, page numbers that appear in bold indicate the main reference for that entry.

Chapter 2

CVODES Installation Procedure

Generally speaking, the installation procedure outlined in §2.1 below will work on commodity LINUX/UNIX systems without modification. Users are still encouraged, however, to carefully read the entire chapter before attempting to install the SUNDIALS suite, in case non-default choices are desired for compilers, compilation options, or the like. In lieu of reading the option list below, the user may invoke the configuration script with the help flag to view a complete listing of available options, which may be done by issuing

```
% ./configure --help
```

from within the `sundials` directory.

In the descriptions below, *build_tree* refers to the directory under which the user wants to build and/or install the SUNDIALS package. By default, the SUNDIALS libraries and header files are installed under the subdirectories *build_tree*`/lib` and *build_tree*`/include`, respectively. Also, *source_tree* refers to the directory where the SUNDIALS source code is located. The chosen *build_tree* may be different from the *source_tree*, thus allowing for multiple installations of the SUNDIALS suite with different configuration options.

Concerning the installation procedure outlined below, after invoking the `tar` command with the appropriate options, the contents of the SUNDIALS archive (or the *source_tree*) will be extracted to a directory named `sundials`. Since the name of the extracted directory is not version-specific it is recommended that the user refrain from extracting the archive to a directory containing a previous version/release of the SUNDIALS suite. If the user is only upgrading and the previous installation of SUNDIALS is not needed, then the user may remove the previous installation by issuing

```
% rm -rf sundials
```

from a shell command prompt.

Even though the installation procedure given below presupposes that the user will use the default vector modules supplied with the distribution, using the SUNDIALS suite with a user-supplied vector module normally will not require any changes to the build procedure.

2.1 Installation steps

To install the SUNDIALS suite, given a downloaded file named *sundials_file*`.tar.gz`, issue the following commands from a shell command prompt, while within the directory where *source_tree* is to be located. The names of installed libraries and header files are listed in Table 2.1 for reference. (For brevity, the corresponding `.c` files are not listed.) Regarding the file extension *.lib* appearing in Table 2.1, shared libraries generally have an extension of `.so` and static libraries have an extension of `.a`. (See *Options for library support* for additional details.)

1. `gunzip` *sundials_file*`.tar.gz`

2. `tar -xf` *sundials_file*`.tar` [creates `sundials` directory]
3. `cd` *build_tree*
4. *path_to_source_tree*`/configure` *options* [options can be absent]
5. `make`
6. `make install`
7. `make examples`
8. If system storage space conservation is a priority, then issue
   `% make clean`
   and/or
   `% make examples_clean`
   from a shell command prompt to remove unneeded object files.

2.2 Configuration options

The installation procedure given above will generally work without modification; however, if the system includes multiple MPI implementations, then certain configure script-related options may be used to indicate which MPI implementation should be used. Also, if the user wants to use non-default language compilers, then, again, the necessary shell environment variables must be appropriately redefined. The remainder of this section provides explanations of available configure script options.

General options

`--prefix=PREFIX`

Location for architecture-independent files.

Default: `PREFIX=`*build_tree*

`--includedir=DIR`

Alternate location for installation of header files.

Default: `DIR=PREFIX/include`

`--libdir=DIR`

Alternate location for installation of libraries.

Default: `DIR=PREFIX/lib`

`--disable-examples`

All available example programs are automatically built unless this option is given. The example executables are stored under the following subdirectories of the associated solver:

build_tree/*solver*/`examples_ser` : serial C examples

build_tree/*solver*/`examples_par` : parallel C examples (MPI-enabled)

build_tree/*solver*/`fcmix/examples_ser` : serial Fortran examples

build_tree/*solver*/`fcmix/examples_par` : parallel Fortran examples (MPI-enabled)

Note: Some of these subdirectories may not exist depending upon the solver and/or the configuration options given.

Table 2.1: SUNDIALS libraries and header files

Module	Libraries	Header files
SHARED	libsundials_shared.*lib*	sundialstypes.h sundialsmath.h sundials_config.h dense.h smalldense.h band.h spgmr.h iterative.h nvector.h
NVECTOR_SERIAL	libsundials_nvecserial.*lib* libsundials_fnvecserial.a	nvector_serial.h
NVECTOR_PARALLEL	libsundials_nvecparallel.*lib* libsundials_fnvecparallel.a	nvector_parallel.h
CVODE	libsundials_cvode.*lib* libsundials_fcvode.a	cvode.h cvdense.h cvband.h cvdiag.h cvspgmr.h cvbandpre.h cvbbdpre.h
CVODES	libsundials_cvodes.*lib*	cvodes.h cvodea.h cvdense.h cvband.h cvdiag.h cvspgmr.h cvbandpre.h cvbbdpre.h
IDA	libsundials_ida.*lib*	ida.h idadense.h idaband.h idaspgmr.h idabbdpre.h
KINSOL	libsundials_kinsol.*lib* libsundials_fkinsol.a	kinsol.h kinspgmr.h kinbbdpre.h

--disable-*solver*

Although each existing solver module is built by default, support for a given solver can be explicitly disabled using this option. The valid values for *solver* are: cvode, cvodes, ida, and kinsol.

--with-cppflags=ARG

Specify additional C preprocessor flags (e.g., ARG=-I<include_dir> if necessary header files are located in nonstandard locations).

--with-cflags=ARG

Specify additional C compilation flags.

--with-ldflags=ARG

Specify additional linker flags (e.g., ARG=-L<lib_dir> if required libraries are located in nonstandard locations).

--with-libs=ARG

Specify additional libraries to be used (e.g., ARG=-l<foo> to link with the library named libfoo.a or libfoo.so).

--with-precision=ARG

By default, SUNDIALS will define a real number (internally referred to as realtype) to be a double-precision floating-point numeric data type (double C-type); however, this option may be used to build SUNDIALS with realtype alternatively defined as a single-precision floating-point numeric data type (float C-type) if ARG=single, or as a long double C-type if ARG=extended.

Default: ARG=double

Options for Fortran support

--disable-f77

Using this option will disable all FORTRAN support. The FCVODE, FKINSOL and FNVECTOR modules will not be built regardless of availability.

--with-fflags=ARG

Specify additional FORTRAN compilation flags.

The configuration script will attempt to automatically determine the function name mangling scheme required by the specified FORTRAN compiler, but the following two options may be used to override the default behavior.

--with-f77underscore=ARG

This option pertains to the FKINSOL, FCVODE and FNVECTOR FORTRAN-C interface modules and is used to specify the number of underscores to append to function names so FORTRAN routines can properly link with the associated SUNDIALS libraries. Valid values for ARG are: none, one and two.

Default: ARG=one

--with-f77case=ARG

Use this option to specify whether the external names of the FKINSOL, FCVODE and FNVECTOR FORTRAN-C interface functions should be lowercase or uppercase so FORTRAN routines can properly link with the associated SUNDIALS libraries. Valid values for ARG are: lower and upper.

Default: ARG=lower

Options for MPI support

The following configuration options are only applicable to the parallel SUNDIALS packages:

`--disable-mpi`

Using this option will completely disable MPI support.

`--with-mpicc=ARG`

`--with-mpif77=ARG`

By default, the configuration utility script will use the MPI compiler scripts named `mpicc` and `mpif77` to compile the parallelized SUNDIALS subroutines; however, for reasons of compatibility, different executable names may be specified via the above options. Also, `ARG=no` can be used to disable the use of MPI compiler scripts, thus causing the serial C and FORTRAN compilers to be used to compile the parallelized SUNDIALS functions and examples.

`--with-mpi-root=MPIDIR`

This option may be used to specify which MPI implementation should be used. The SUNDIALS configuration script will automatically check under the subdirectories `MPIDIR/include` and `MPIDIR/lib` for the necessary header files and libraries. The subdirectory `MPIDIR/bin` will also be searched for the C and FORTRAN MPI compiler scripts, unless the user uses `--with-mpicc=no` or `--with-mpif77=no`.

`--with-mpi-incdir=INCDIR`

`--with-mpi-libdir=LIBDIR`

`--with-mpi-libs=LIBS`

These options may be used if the user would prefer not to use a preexisting MPI compiler script, but instead would rather use a serial complier and provide the flags necessary to compile the MPI-aware subroutines in SUNDIALS.

Often an MPI implementation will have unique library names and so it may be necessary to specify the appropriate libraries to use (e.g., `LIBS=-lmpich`).

Default: `INCDIR=MPIDIR/include`, `LIBDIR=MPIDIR/lib` and `LIBS=-lmpi`

`--with-mpi-flags=ARG`

Specify additional MPI-specific flags.

Options for library support

By default, only static libraries are built, but the following option may be used to build shared libraries on supported platforms.

`--enable-shared`

Using this particular option will result in both static and shared versions of the available SUNDIALS libraries being built if the system supports shared libraries. To build only shared libraries also specify `--disable-static`.

Note: The FCVODE and FKINSOL libraries can only be built as static libraries because they contain references to externally defined symbols, namely user-supplied FORTRAN subroutines. Although the FORTRAN interfaces to the serial and parallel implementations of the supplied NVECTOR module do not contain any unresolvable external symbols, the libraries are still built as static libraries for the purpose of consistency.

Options for cross-compilation

If the SUNDIALS suite will be cross-compiled (meaning the build procedure will not be completed on the actual destination system, but rather on an alternate system with a different architecture) then the following two options should be used:

`--build=BUILD`

This particular option is used to specify the canonical system/platform name for the build system.

`--host=HOST`

If cross-compiling, then the user must use this option to specify the canonical system/platform name for the destination system.

Environment variables

The following environment variables can be locally (re)defined for use during the configuration of SUNDIALS. See the next section for illustrations of these.

`CC`

`F77`

Since the configuration script uses the first C and FORTRAN compilers found in the current executable search path, then each relevant shell variable (`CC` and `F77`) must be locally (re)defined in order to use a different compiler. For example, to use `xcc` (executable name of chosen compiler) as the C language compiler, use `CC=xcc` in the configure step.

`CFLAGS`

`FFLAGS`

Use these environment variables to override the default C and FORTRAN compilation flags.

2.3 Configuration examples

The following examples are meant to help demonstrate proper usage of the configure options:

```
% configure CC=gcc F77=g77 --with-cflags=-g3 --with-fflags=-g3 \
            --with-mpicc=/usr/apps/mpich/1.2.4/bin/mpicc \
            --with-mpif77=/usr/apps/mpich/1.2.4/bin/mpif77
```

The above example builds SUNDIALS using `gcc` as the serial C compiler, `g77` as the serial FORTRAN compiler, `mpicc` as the parallel C compiler, `mpif77` as the parallel FORTRAN compiler, and appends the `-g3` compilaton flag to the list of default flags.

```
% configure CC=gcc --disable-examples --with-mpicc=no \
            --with-mpi-root=/usr/apps/mpich/1.2.4 \
            --with-mpi-libs=-lmpich
```

This example again builds SUNDIALS using `gcc` as the serial C compiler, but the `--with-mpicc=no` option explicitly disables the use of the corresponding MPI compiler script. In addition, since the `--with-mpi-root` option is given, the compilation flags `-I/usr/apps/mpich/1.2.4/include` and `-L/usr/apps/mpich/1.2.4/lib` are passed to `gcc` when compiling the MPI-enabled functions. The `--disable-examples` option disables the examples (which means a FORTRAN compiler is not required). The `--with-mpi-libs` option is still needed so that the configure script can check if `gcc` can link with the appropriate MPI library as `-lmpi` is the internal default.

Chapter 3

Mathematical Considerations

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

$$\dot{y} = f(t, y), \quad y(t_0) = y_0, \tag{3.1}$$

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

$$\begin{aligned} \dot{y} &= f(t,\, y,\, p) \\ y(t_0) &= y_0(p)\,, \end{aligned} \tag{3.2}$$

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.

3.1 IVP solution

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

$$\sum_{i=0}^{K_1} \alpha_{n,i} y^{n-i} + h_n \sum_{i=0}^{K_2} \beta_{n,i} \dot{y}^{n-i} = 0\,. \tag{3.3}$$

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 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 [4] and [20].

For either choice of formula, the nonlinear system

$$G(y_n) \equiv y^n - h_n \beta_{n,0} f(t_n, y^n) - a_n = 0\,, \tag{3.4}$$

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

$$M[y^{n(m+1)} - y^{n(m)}] = -G(y^{n(m)}), \tag{3.5}$$

in which

$$M \approx I - \gamma J, \quad J = \partial f / \partial y, \quad \text{and} \quad \gamma = h_n \beta_{n,0}. \tag{3.6}$$

The initial guess for the iteration is a predicted value $y^{n(0)}$ computed explicitly from the available history data. For the Newton corrections, CVODES provides a choice of four methods:

- a dense direct solver (serial version only),
- a band direct solver (serial version only),
- a diagonal approximate Jacobian solver, or
- SPGMR = Scaled Preconditioned GMRES, without restarts.

For large stiff systems, where direct methods are not feasible, the combination of a BDF integrator and a preconditioned GMRES algorithm 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 [2].

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 weights used are based on the current solution and on the relative and absolute tolerances input by the user, namely

$$W_i = \text{RTOL} \cdot |y_i| + \text{ATOL}_i. \tag{3.7}$$

Because 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.

For the direct solvers (dense, band and diagonal), the iteration is a modified Newton iteration since the iteration matrix M is fixed throughout the nonlinear iterations. However, for SPGMR, 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 (for SPGMR) 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 re-evaluate 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 $\|y^{n(m)} - y^{n(0)}\| \le \epsilon$. Letting y^n denote the exact solution of (3.4), 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 SPGMR 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 in SPGMR 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 (3.7). 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.

In the case of SPGMR, 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

$$Jv = [f(t, y + \sigma v) - f(t, y)]/\sigma . \tag{3.8}$$

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

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

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

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

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 [1, 3, 14]. They are also summarized in [15].

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 $t = t_{\text{stop}}$.

3.2 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 (3.2). 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 in short *sensitivity equations*):

$$\dot{s}_i = \frac{\partial f}{\partial y} s_i + \frac{\partial f}{\partial p_i}\,, \quad s_i(t_0) = \frac{\partial y_0(p)}{\partial p_i}\,, \tag{3.9}$$

obtained by applying the chain rule of differentiation to the original ODEs (3.2).

When performing forward sensitivity analysis, CVODES carries out the time integration of the combined system, (3.2) and (3.9), 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 the desired sensitivities ($N_s \le N_p$). 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 (3.6).

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.

3.2.1 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 $\hat{y} = [y, s_1, \dots, s_{N_s}]$.

- *Staggered Direct*

  In this approach [7], the nonlinear system (3.4) is first solved and, once an acceptable numerical solution is obtained, the sensitivity variables at the new step are found by directly solving (3.9) 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 (3.6), it must be updated and factored at every step of the integration, in contrast to 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 [21]. 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 [22], the discretization is applied simultaneously to both the original equations (3.2) and the sensitivity systems (3.9) 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 $\hat{f} = [f(t,y,p), \dots, (\partial f/\partial y)(t,y,p)s_i + (\partial f/\partial p_i)(t,y,p), \dots]$, 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 (3.5) by solving the corrector equation

  $$\hat{M}[\hat{y}_{n(m+1)} - \hat{y}_{n(m)}] = -\hat{G}(\hat{y}_{n(m)}) \tag{3.10}$$

  at each iteration, where

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

M is defined as in (3.6), and $J_i = (\partial/\partial y)\,[(\partial f/\partial y)s_i + (\partial f/\partial p_i)]$. It can be shown that 2-step quadratic convergence can be attained by using only the block-diagonal portion of $\hat{M}$ in the corrector equation (3.10). 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 (3.10) to update the sensitivity portions of the residual $\hat{G}$.

- *Staggered corrector*

  In this approach [10], as in the staggered direct method, the nonlinear system (3.4) is solved first using the Newton iteration (3.5). Then a separate Newton iteration is used to solve the sensitivity system (3.9):

$$M[s_i^{n(m+1)} - s_i^{n(m)}] = \\ - \left[s_i^{n(m)} - \gamma \left(\frac{\partial f}{\partial y}(t_n, y^n, p) s_i^{n(m)} + \frac{\partial f}{\partial p_i}(t_n, y^n, p) \right) - a_{i,n} \right], \quad (3.11)$$

  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 (3.5) has converged. Note also that Jacobian-related data can be reused at all iterations (3.11) 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 (3.11) 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 the SPGMR linear solver, effectively results in a staggered direct method. Indeed, SPGMR 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 (3.11) will theoretically converge after one iteration.

3.2.2 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.

3.2.3 Evaluation of the sensitivity right-hand side

There are several methods for evaluating the right-hand side of the sensitivity systems (3.9): 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 or complex-step approximation, and future versions will provide these capabilities. 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 (3.9) can be evaluated separately:

$$\frac{\partial f}{\partial y}s_i \approx \frac{f(t, y + \sigma_y s_i, p) - f(t, y - \sigma_y s_i, p)}{2\,\sigma_y}\,, \tag{3.12}$$

$$\frac{\partial f}{\partial p_i} \approx \frac{f(t, y, p + \sigma_i e_i) - f(t, y, p - \sigma_i e_i)}{2\,\sigma_i}\,, \tag{3.12'}$$

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

simultaneously:

$$\frac{\partial f}{\partial y}s_i + \frac{\partial f}{\partial p_i} \approx \frac{f(t, y + \sigma s_i, p + \sigma e_i) - f(t, y - \sigma s_i, p - \sigma e_i)}{2\,\sigma}\,, \tag{3.13}$$

$$\sigma = \min(\sigma_i, \sigma_y)\,,$$

or by adaptively switching between (3.12)+(3.12') and (3.13), depending on the relative size of the estimated finite difference increments σ_i and σ_y.

These procedures for choosing the perturbations (δ_i, δ_y, δ) and switching ($\rho_{\max}$) between finite difference and directional derivative formulas have also been implemented for first-order 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

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

can be used. In CVODES, the default value of $\rho_{\max} = 0$ indicates the use of the second-order centered directional derivative formula (3.13) 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.

3.3 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 (3.2), we wish to evaluate the gradient dG/dp of

$$G(p) = \int_{t_0}^{t_{\text{f}}} g(t, y, p)dt\,, \tag{3.14}$$

or, alternatively, the gradient dg/dp of the function $g(t, x, p)$ at time t_{f}. 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 [6]. Introducing a Lagrange multiplier λ, we form the augmented objective function

$$I(p) = G(p) - \int_{t_0}^{t_1} \lambda^* \left(\dot{y} - f(t,y,p) \right) dt \,, \tag{3.15}$$

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

$$\frac{dG}{dp} = \frac{dI}{dp} = \int_{t_0}^{t_1} (g_p + g_y s) dt - \int_{t_0}^{t_1} \lambda^* \left(\dot{s} - f_y s - f_p \right) dt \,, \tag{3.16}$$

where subscripts on functions such as 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 selecting λ such that

$$\begin{aligned} \dot{\lambda} &= -\left(\frac{\partial f}{\partial y}\right)^* \lambda - \left(\frac{\partial g}{\partial y}\right)^* \\ \lambda(t_1) &= 0 \,, \end{aligned} \tag{3.17}$$

the gradient of G with respect to p is nothing but

$$\frac{dG}{dp} = \lambda^*(t_0) s(t_0) + \int_{t_0}^{t_1} \left(g_p + \lambda^* f_p \right) dt \,. \tag{3.18}$$

The gradient of $g(t_1, y, p)$ with respect to p can be then obtained by using the Leibnitz differentiation rule. Indeed, from (3.14),

$$\frac{dg}{dp}(t_1) = \frac{d}{dt_1} \frac{dG}{dp}$$

and therefore, taking into account that dG/dp in (3.18) depends on t_1 both through the upper integration limit and through λ and that $\lambda(t_1) = 0$,

$$\frac{dg}{dp}(t_1) = \mu^*(t_0) s(t_0) + g_p(t_1) + \int_{t_0}^{t_1} \mu^* f_p dt \,, \tag{3.19}$$

where μ is the sensitivity of λ with respect to the final integration limit and thus satisfies the following equation, obtained by taking the total derivative with respect to t_1 of (3.17):

$$\begin{aligned} \dot{\mu} &= -\left(\frac{\partial f}{\partial y}\right)^* \mu \\ \mu(t_1) &= \left(\frac{\partial g}{\partial y}\right)^*_{t=t_1} \,. \end{aligned} \tag{3.20}$$

The final condition on $\mu(t_1)$ follows from $(\partial\lambda/\partial t) + (\partial\lambda/\partial t_1) = 0$ at t_1, and therefore, $\mu(t_1) = -\dot{\lambda}(t_1)$.

The first thing to notice about the adjoint system (3.17) is that there is no explicit specification of the parameters p; this implies that, once the solution λ is found, the formula (3.18) 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 (3.20) and the formula (3.19) for gradients of $g(t_1, y, p)$. The second important remark is that the adjoint systems (3.17) and (3.20) are terminal value problems which depend on the solution $y(t)$ of the original IVP (3.2). Therefore, a procedure is needed for providing the states y obtained during a forward integration phase of (3.2) to CVODES during the backward integration phase of (3.17) or (3.20). The approach adopted in CVODES, based on *checkpointing*, is described below.

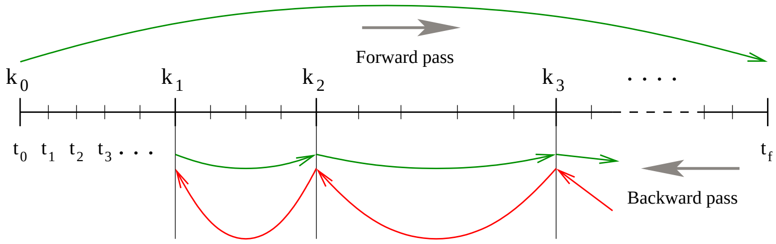


Figure 3.1: Illustration of the checkpointing algorithm for generation of the forward solution during the integration of the adjoint system.

3.3.1 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. Therefore, CVODES implements a cubic Hermite interpolation algorithm. 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.

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 $(y, \dot{y})$ 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_1 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 N_d data pairs $(y, \dot{y})$ are generated and stored in memory for interpolation. (see Fig. 3.1).

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, data pairs $(y, \dot{y})$ 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 (3.2)), 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 (3.14).

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 (3.2), including adjoint systems (3.17) or (3.20), as well as any other quadrature ODEs that may be needed in evaluating the integrals in (3.18) or (3.19). 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.

3.4 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 [12]. 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 [13], 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.

3.5 Rootfinding

The CVODES solver has been augmented to include a rootfinding feature. This means that, while integrating the Initial Value Problem (3.1), 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 [11]. 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 $t + \tau$ for a small (near roundoff level) increment τ, slightly further in the direction of integration, and if any $g_i(t + \tau) = 0$ 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 $\tau/2$.

Chapter 4

Code Organization

4.1 SUNDIALS organization

The family of solvers referred to as SUNDIALS consists of the solvers CVODE (for ODE systems), KINSOL (for nonlinear algebraic systems), and IDA (for differential-algebraic systems). In addition, variants of these which also do sensitivity analysis calculations are available or in development. CVODES, an extension of CVODE that provides both forward and adjoint sensitivity capabilities is available, while IDAS is currently in development.

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. 4.1). The following is a list of the solver packages presently available:

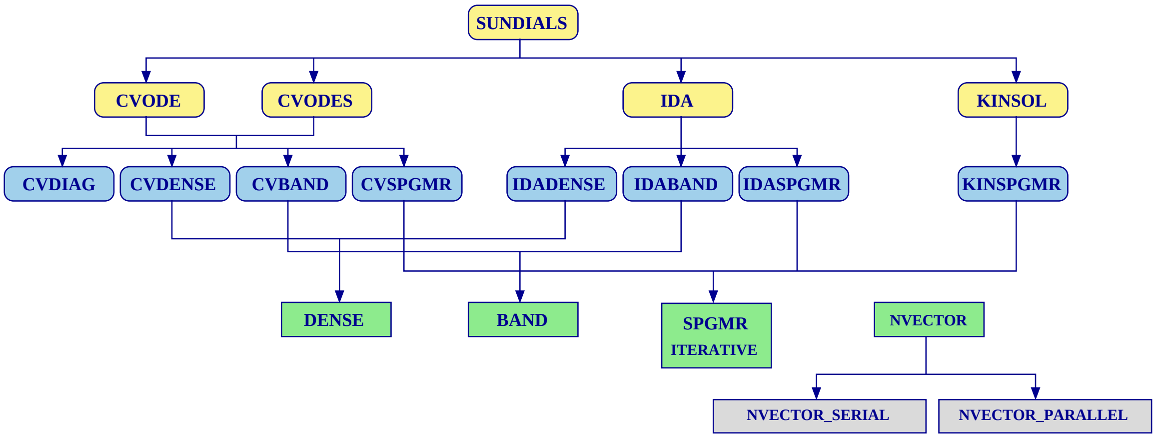
- CVODE, a solver for stiff and nonstiff ODEs $dy/dt = f(t, y)$;
- CVODES, a solver for stiff and nonstiff ODEs $dy/dt = f(t, y, p)$ with sensitivity analysis capabilities;
- KINSOL, a solver for nonlinear algebraic systems $F(u) = 0$;
- IDA, a solver for differential-algebraic systems $F(t, y, y') = 0$.

4.2 CVODES organization

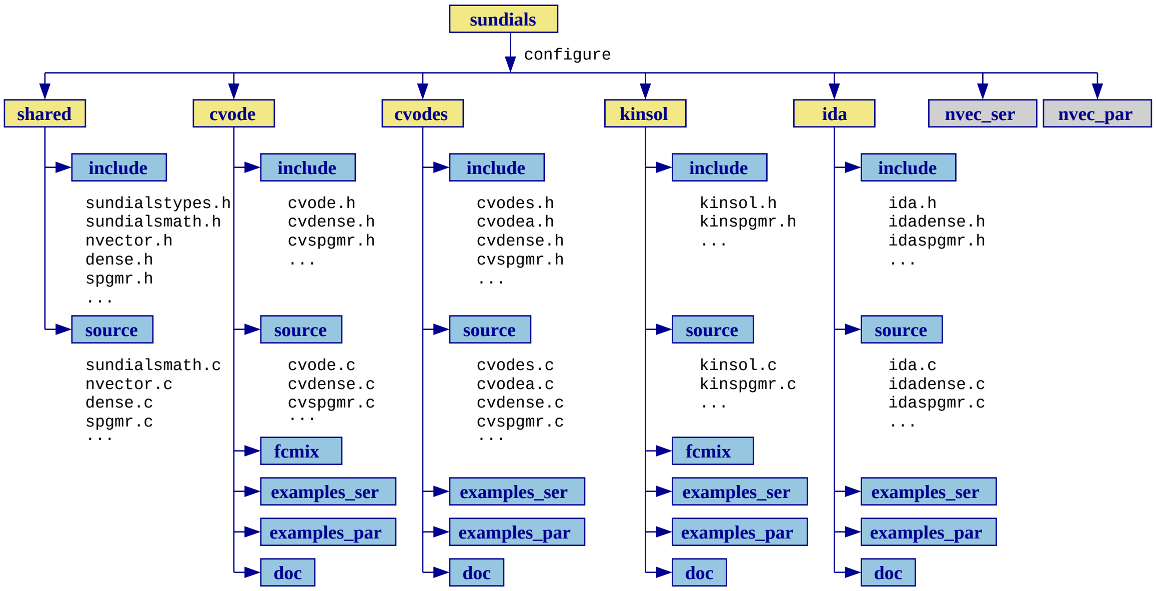
The CVODES package is written in ANSI C. The following summarizes the basic structure of the package, although knowledge of this structure is not necessary for its use.

The overall organization of the CVODES package is shown in Figure 4.2. 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` 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` and `CV_STAGGERED1` (see §3.2 and §6.2.1). The CVODES package includes an algorithm



(a) High-level diagram



(b) Directory structure

Figure 4.1: Organization of the SUNDIALS suite

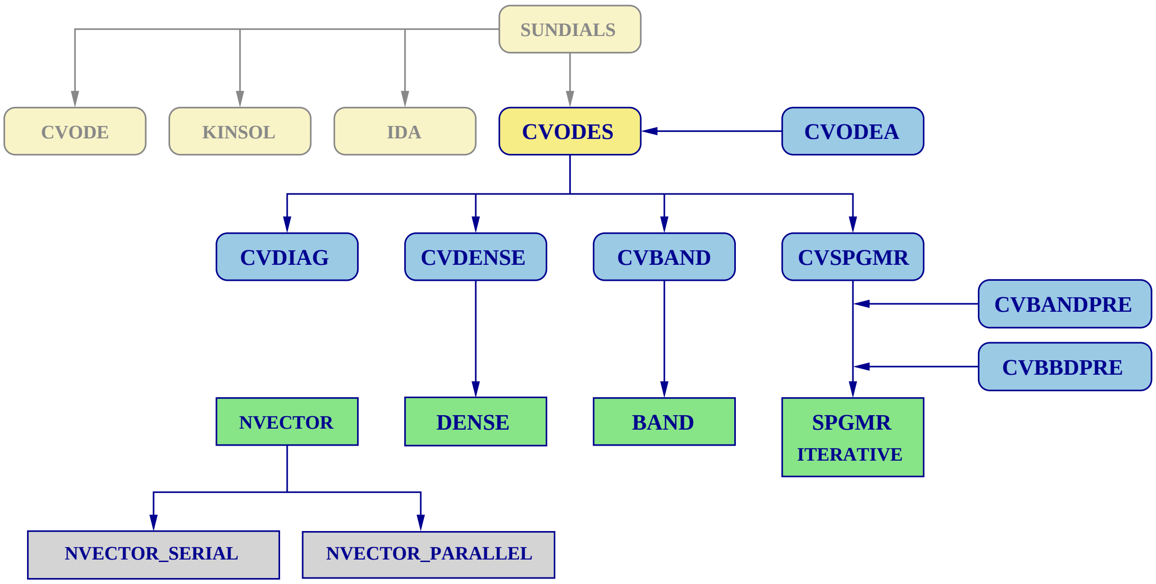


Figure 4.2: Overall structure of the CVODES package. Modules specific to CVODES are distinguished by rounded boxes, while generic solver and auxiliary modules are in rectangular boxes.

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 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 four CVODES linear solver modules:

- CVDENSE: LU factorization and backsolving with dense matrices;
- CVBAND: LU factorization and backsolving with banded matrices;
- CVDIAG: an internally generated diagonal approximation to the Jacobian;
- CVSPGMR: scaled preconditioned GMRES method.

This set of linear solver modules is intended to be expanded in the future as new algorithms are developed.

In the case of the direct CVDENSE and CVBAND methods, the package includes an algorithm for the approximation of the Jacobian by difference quotients, but the user also has the option of supplying the Jacobian (or an approximation to it) directly. In the case of the iterative CVSPGMR method, the package includes an algorithm for the approximation of the product between the Jacobian matrix and a vector of appropriate length by difference quotients. Again, the user has the option of providing a routine for this operation. In the case of CVSPGMR, the preconditioner must be supplied by the user in two parts: setup (preprocessing of Jacobian data) and solve. While there is no default choice of preconditioner analogous to the difference quotient approximation in the direct case, the references [2] and [3], together with the example programs included with CVODES, offer considerable assistance in building preconditioners.

Each CVODES linear solver module consists of four routines devoted to (1) memory allocation and initialization, (2) setup of the matrix data involved, (3) solution of the system, and (4) freeing of memory. The setup and solution phases are separate because the evaluation of Jacobians and preconditioners is done only periodically during the integration, and only as required to achieve convergence. The call list within the central 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. Each of the modules CVDENSE, CVBAND and CVSPGMR is a set of interface routines built on top of a generic solver module, named DENSE, BAND and SPGMR, respectively. The interfaces deal with the use of these methods in the CVODES context, whereas the generic solver is independent of the context. While the generic solvers here were generated with SUNDIALS in mind, our intention is that they be usable in other applications 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. The first one, CVBANDPRE, is intended to be used with NVECTOR_SERIAL and provides banded difference quotient Jacobian-based preconditioner and solver routines for use with CVSPGMR. 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 5

Using CVODES for IVP Solution

This chapter is concerned with the use of CVODES for the solution of 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 [18]. The listings of the sample programs in the companion document [16] may also be helpful. Those codes may be used as templates 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 or direct band linear solvers since these linear solver modules need to form the complete system Jacobian. The following CVODES modules can only be used with NVECTOR_SERIAL: CVDENSE, CVBAND and CVBANDPRE. 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 Chapter 11.

5.1 Access to libraries and header files

At this point, it is assumed that the installation of CVODES, following the procedure described in Chapter 2, has been completed successfully.

Regardless of where the user's application program resides, its associated compilation and load commands must make reference to the locations of the libraries and header files required by CVODES. In terms of the directory *build_tree* defined in Chapter 2, the relevant library files are

- *build_tree*`/lib/libsundials_cvodes.`*lib*,
- *build_tree*`/lib/libsundials_shared.`*lib*, and
- *build_tree*`/lib/libsundials_nvec*.`*lib* (up to two files)

where the file extension *.lib* is typically `.so` for shared libraries and `.a` for static libraries. All relevant header files are located under the subdirectory

- *build_tree*`/include`

5.2 Data Types

The `sundialstypes.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 §2.2).

Additionally, based on the current precision, `sundialstypes.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 the typedef for `realtype` matches this choice). 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 §2.2).

5.3 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 `sundialstypes.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 §8 for details). For the two 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-based) implementation, NVECTOR_PARALLEL.

Note that both these files in turn include the header file `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:

- `cvdense.h`, which is used with the dense direct linear solver in the context of CVODES. This in turn includes a header file (`dense.h`) which defines the `DenseMat` type and corresponding accessor macros;
- `cvband.h`, which is used with the band direct linear solver in the context of CVODES. This in turn includes a header file (`band.h`) which defines the `BandMat` type and corresponding accessor macros;
- `cvdiag.h`, which is used with the diagonal linear solver in the context of CVODES;

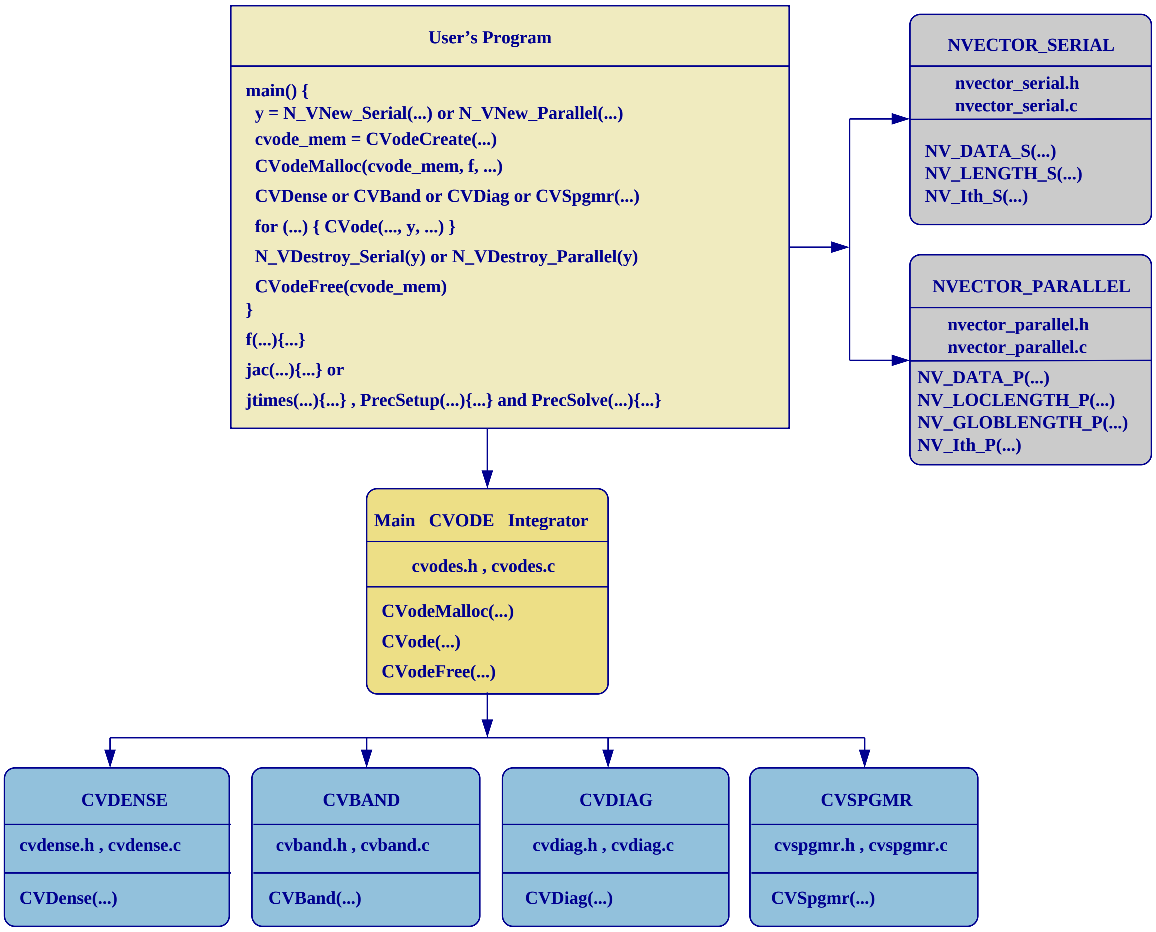


Figure 5.1: Diagram of user program and CVODES package for integration of IVP

- `cvspgmr.h`, which is used with the Krylov solver SPGMR in the context of CVODES. This in turn includes a header file (`iterative.h`) which enumerates the kind of preconditioning and the available choices for the Gram-Schmidt process.

Other headers may be needed, depending upon the choice of preconditioner, etc. In one of the examples in [17], preconditioning is done with a block-diagonal matrix. For this, the header `smalldense.h` is included.

5.4 A skeleton of the user's main program

A high-level overview of the combined user program and CVODES package is shown in Figure 5.1. 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 with [**P**] correspond to NVECTOR_PARALLEL, while steps marked with [**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**] Set `N`, the problem size N.

[**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.

3. **Set vector of initial values**

   To set the vector `y0` of initial values, use the appropriate functions defined by a particular NVECTOR implementation. If a `realtype` array `ydata` containing the initial values of y already exists, then make the call:

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

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

   Otherwise, make the call:

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

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

   and load initial values into the structure defined by:

   [**S**] `NV_DATA_S(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 §5.5.1 for details.

5. **Allocate internal memory**

   Call `CVodeMalloc(...);` to provide required problem specifications, allocate internal memory for CVODES, and initialize CVODES. `CVodeMalloc` returns a flag, the value of which indicates either success or an illegal argument value. See §5.5.1 for details.

6. **Set optional inputs**

   Call `CVodeSet*` functions to change any optional inputs that control the behavior of CVODES from their default values. See §5.5.4 for details.

7. **Attach linear solver module**

   If Newton iteration is chosen, initialize the linear solver module with one of the following calls (for details see §5.5.2):

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

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

   `ier = CVDiag(...);`

   `ier = CVSpgmr(...);`

8. **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 §5.5.4 for details.

9. **Specify rootfinding problem**

Optionally, call `CVodeRootInit` to initialize a rootfinding problem to be solved during the integration of the ODE system. See §5.8.1 for details.

10. **Advance solution in time**

    For each point at which output is desired, call `ier = CVode(cvode_mem, tout, yout, &tret, itask);` Set `itask` to specify the return mode. The vector `y` (which can be the same as the vector `y0` above) will contain $y(t)$. See §5.5.3 for details.

11. **Get optional outputs**

    Call `CV*Get*` functions to obtain optional output. See §5.5.6 and §5.8.1 for details.

12. **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**] `NV_Destroy_Serial(y);`

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

13. **Free solver memory**

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

14. **[P] Finalize MPI**

    Call `MPI_Finalize();` to terminate MPI.

5.5 User-callable functions for IVP solution

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 §5.5.4, 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 §5.4 for the correct order of these calls. Calls related to rootfinding are described in §5.8.

5.5.1 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 *`). If an error occurred, `CVodeCreate` prints an error message to `stderr` and returns `NULL`.

`CVodeMalloc`

Call `flag = CVodeMalloc(cvode_mem, f, t0, y0, itol, reltol, abstol);`

Description The function `CVodeMalloc` 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 f in the ODE. This function has the form `f(t, y, ydot, f_data)` (for full details see §5.6.1).
- `t0` (`realtype`) is the initial value of t.
- `y0` (`N_Vector`) is the initial value of y.
- `itol` (`int`) is one of `CV_SS`, `CV_SV`, or `CV_WF`, where `itol`=`SS` indicates scalar relative error tolerance and scalar absolute error tolerance, while `itol`=`CV_SV` indicates scalar relative error tolerance and vector absolute error tolerance. The latter choice is important when the absolute error tolerance needs to be different for each component of the ODE. If `itol`=`CV_WF`, the arguments `reltol` and `abstol` are ignored and the user is expected to provide a function to evaluate the error weight vector W from (3.7). See `CVodeSetEwtFn` in §5.5.4.
- `reltol` (`realtype`) is the relative error tolerance.
- `abstol` (`void *`) is a pointer to the absolute error tolerance. If `itol`=`CV_SS`, `abstol` must be a pointer to a `realtype` variable. If `itol`=`CV_SV`, `abstol` must be an `N_Vector` variable.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeMalloc` 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 `CVodeMalloc` has an illegal value.

Notes If an error occurred, `CVodeMalloc` also prints an error message to the file specified by the optional input `errfp`.

The tolerance values in `reltol` and `abstol` may be changed between calls to `CVode` (see `CVodeSetTolerances` in §5.5.4).

! It is the user's responsibility to provide compatible `itol` and `abstol` arguments.

`CVodeFree`

Call `CVodeFree(cvode_mem);`

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

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

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

5.5.2 Linear solver specification functions

As previously explained, Newton iteration requires the solution of linear systems of the form (3.5). There are four CVODES linear solvers currently available for this task: CVDENSE, CVBAND, CVDIAG, and CVSPGMR. The first three are direct solvers and derive their names from the type of approximation used for the Jacobian $J = \partial f/\partial y$. CVDENSE, CVBAND and CVDIAG work with dense, banded, and diagonal approximations to J, respectively. The fourth CVODES linear solver, CVSPGMR, is an iterative solver. The SPGMR in the name indicates that it uses a scaled preconditioned GMRES method.

To specify a CVODES linear solver, after the call to `CVodeCreate` but before any calls to `CVode`, the user's program must call `CVDense`, `CVBand`, `CVDiag`, or `CVSpgmr`, 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 in 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 and SPGMR, are described separately in Chapter 10.

`CVDense`

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

Description The function `CVDense` selects the CVDENSE linear solver.

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

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

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

- `CVDENSE_SUCCESS` The CVDENSE initialization was successful.
- `CVDENSE_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVDENSE_ILL_INPUT` The CVDENSE solver is not compatible with the current NVECTOR module.
- `CVDENSE_MEM_FAIL` A memory allocation request failed.

Notes The CVDENSE linear solver may not be compatible with a particular implementation of the NVECTOR module. Of the two NVECTOR modules provided with SUNDIALS, only NVECTOR_SERIAL is compatible.

`CVBand`

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

Description The function `CVBand` selects the CVBAND linear solver.

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `N` (`long int`) problem dimension.
- `mupper` (`long int`) upper half-bandwidth of the problem Jacobian (or of the approximation of it).
- `mlower` (`long int`) lower half-bandwidth of the problem Jacobian (or of the approximation of it).

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

- `CVBAND_SUCCESS` The CVBAND initialization was successful.
- `CVBAND_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVBAND_ILL_INPUT` The CVBAND solver is not compatible with the current NVECTOR module, or one of the Jacobian half-bandwidths is outside of its valid range ($0 \ldots$ `N`-1).
- `CVBAND_MEM_FAIL` A memory allocation request failed.

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

`CVDiag`

Call: `flag = CVDiag(cvode_mem);`

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

The user's main function must include the `cvdiag.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.

`CVSpgmr`

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

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

The user's main function must include the `cvspgmr.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 `CVSPGMR_MAXL = 5`.

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

- `CVSPGMR_SUCCESS` The CVSPGMR initialization was successful.
- `CVSPGMR_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVSPGMR_ILL_INPUT` The preconditioner type `pretype` is not valid.
- `CVSPGMR_MEM_FAIL` A memory allocation request failed.

Notes: The CVSPGMR solver uses a scaled preconditioned GMRES iterative method to solve the linear system (3.5).

With the SPGMR method, preconditioning can be done on the left only, on the right only, on both the left and the right, or not at all. For a given preconditioner matrix, the merits of left vs. right preconditioning are generally unclear, and so 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 SPGMR 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. For a specification of the preconditioner, see §5.5.4 and §5.6.

If preconditioning is done, then the user must supply functions defining the 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 (3.6).

5.5.3 CVODE solver function

This is the central step in the solution process - the call to perform the integration of the IVP.

`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.
- `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. The `CV_NORMAL_TSTOP` and `CV_ONE_STEP_TSTOP` modes are similar to `CV_NORMAL` and `CV_ONE_STEP`, respectively, except that the integration never proceeds past the value `tstop` (specified through the function `CVodeSetStopTime`).

Return value On return, `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 §5.5.4).
- `CV_ROOT_RETURN` `CVode` succeeded and found one or more roots. If `nrtfn` > 1, call `CVodeGetRootInfo` to see which g_i were found to have a root. See §5.8 for more information.
- `CV_MEM_NULL` The `cvode_mem` argument was `NULL`.
- `CV_NO_MALLOC` The CVODES memory block was not allocated by a call to `CVodeMalloc`.
- `CV_ILL_INPUT` One of the inputs to `CVode` is illegal. This includes the situation where a root of one of the root functions was found both at a point t and also very near t. It also includes the situation where a component of the error weight vector becomes negative during internal time-stepping. The `CV_ILL_INPUT` flag will also be returned if the linear solver initialization function (called by the user after calling `CVodeCreate`) failed to set the linear solver-specific `lsolve` field in `cvode_mem`. In any case, the user should see the printed error message for details.
- `CV_LINIT_FAIL` The linear solver's initialization function failed.
- `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_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.

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

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

All failure return values are negative and so the test ier < 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.

5.5.4 Optional input functions

CVODES provides an extensive set of functions that can be used to change from their default values various optional input parameters that control the behavior of the CVODES solver. Table 5.1 lists all optional input functions in CVODES which are then described in detail in the remainder of this section. For the most casual use of CVODES, the reader can skip to §5.6.

We note that, on error return, all of the optional input functions print an error message to stderr (or to the file pointed to by errfp, if already specified). We also note that all error return values are negative, so the test flag < 0 will catch all errors.

Main solver optional input functions

The calls listed here can be executed in any order.

CVodeSetErrFile

Call flag = CVodeSetErrFile(cvode_mem, errfp);

Description The function CVodeSetErrFile specifies a pointer to the file where all CVODES messages should be directed.

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).

! 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.

Table 5.1: Optional inputs for CVODES, CVDENSE, CVBAND, and CVSPGMR

Optional input	**Function name**	**Default**
CVODE main solver		
Pointer to an error file	`CVodeSetErrFile`	`stderr`
Data for right-hand side function	`CVodeSetFdata`	`NULL`
Maximum order for BDF method	`CVodeSetMaxOrd`	5
Maximum order for Adams method	`CVodeSetMaxOrd`	12
Maximum no. of internal steps before t_{out}	`CVodeSetMaxNumSteps`	500
Maximum no. of warnings for $t_n + h = t_n$	`CVodeSetMaxHnilWarns`	10
Flag to activate stability limit detection	`CVodeSetStabLimDet`	`FALSE`
Initial step size	`CVodeSetInitStep`	estimated
Minimum absolute step size	`CVodeSetMinStep`	0.0
Maximum absolute step size	`CVodeSetMaxStep`	∞
Value of t_{stop}	`CVodeSetStopTime`	∞
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
Data for rootfinding function	`CVodeSetGdata`	`NULL`
Nonlinear iteration type	`CVodeSetIterType`	none
Integration tolerances	`CVodeSetTolerances`	none
Ewt compuation function	`CVodeSetEwtFn`	internal fn.
CVDENSE linear solver		
Dense Jacobian function and data	`CVDenseSetJacFn`	internal DQ, NULL
CVBAND linear solver		
Band Jacobian function and data	`CVBandSetJacFn`	internal DQ, NULL
CVSPGMR linear solver		
Preconditioner functions and data	`CVSpgmrSetPreconditioner`	NULL, NULL, NULL
Jacobian times vector function and data	`CVSpgmrSetJacTimesVecFn`	internal DQ, NULL
Type of Gram-Schmidt orthogonalization	`CVSpgmrSetGSType`	classical GS
Ratio between linear and nonlinear tolerances	`CVSpgmrSetDelt`	0.05
Preconditioning type	`CVSpgmrSetPrecType`	none

`CVodeSetFdata`

Call `flag = CVodeSetFdata(cvode_mem, f_data);`

Description The function `CVodeSetFdata` specifies the user-defined data block `f_data` to be passed to the user-supplied right-hand side function f, and attaches it to the main CVODES memory block.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `f_data` (`void *`) pointer to the user-defined data 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 If `f_data` is not specified, a `NULL` pointer is passed to the f function.

`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.

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 negative 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.

`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`.
- `CV_ILL_INPUT` `mxsteps` is non-positive.

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

`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

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 §3.4 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, `CVsldet`, is called to detect a possible stability limit. If such a limit is detected, then the order is reduced.

`CVodeSetInitStep`

Call `flag = CVodeSetInitStep(cvode_mem, hin);`

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

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

`hin` (`realtype`) value of the initial step size.

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

`CV_SUCCESS` The optional value has been successfully set.

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

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

`CVodeSetMinStep`

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

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

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

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

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

`CV_SUCCESS` The optional value has been successfully set.

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

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

Notes The default value is 0.0.

`CVodeSetMaxStep`

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

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

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

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`$= 0$ 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`.

Notes The default value is ∞.

`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.

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.

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.

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 §3.1).

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `nlscoef` (`realtype`) coefficient in nonlinear convergence test.

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 §5.5.1). This function call is needed only if `iter` is being changed from its value in the prior call to `CVodeCreate`.

`CVodeSetTolerances`

Call `flag = CVodeSetTolerances(cvode_mem, itol, reltol, abstol);`

Description The function `CVodeSetTolerances` resets the integration tolerances.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `itol` (`int`) is either `CV_SS` or `CV_SV`, where `itol = CV_SS` indicates scalar relative error tolerance and scalar absolute error tolerance, while `itol=CV_SV` indicates scalar relative error tolerance and vector absolute error tolerance. The latter choice is important when the absolute error tolerance needs to be different for each component of the ODE.
- `reltol` (`realtype`) the relative error tolerance.
- `abstol` (`void *`) is a pointer to the absolute error tolerance. If `itol=CV_SS`, `abstol` must be a pointer to a `realtype` variable. If `itol=CV_SV`, `abstol` must be an `N_Vector` variable.

Return value The return value `flag` (of type `int`) is one of
- `CV_SUCCESS` The tolerances have been successfully set.
- `CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CV_ILL_INPUT` An input argument has an illegal value.

Notes The integration tolerances are initially specified in the call to `CVodeMalloc` (see §5.5.1). This function call is needed only if the tolerances are being changed from their values beween successiv calls to `CVode`.

It is the user's responsibility to provide compatible `itol` and `abstol` arguments.

It is illegal to call `CVodeSetTolerances` before a call to `CVodeMalloc`.

`CVodeSetEwtFn`

Call `flag = CVodeSetEwtFn(cvode_mem, efun, e_data);`

Description The function `CVodeSetEwtFn` specifies the user-defined function to be used in computing the error weight vector W in (3.7).

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `efun` (`CVEwtFn`) is the C function which defines the `ewt` vector (see §5.6.2).
- `e_data` (`void *`) pointer to user data passed to `efun` every time it is called.

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

Notes This function can be called between successive calls to `CVode`.

If not needed, pass `NULL` for `edata`.

It is illegal to call `CVodeSetEwtFn` before a call to `CVodeMalloc`.

Linear solver optional input functions

The linear solver modules, with one exception, allow for various optional inputs which are described here. The diagonal linear solver module has no optional inputs.

Dense Linear solver. The CVDENSE solver needs a function to compute a dense approximation to the Jacobian matrix $J(t, y)$. This function must be of type `CVDenseJacFn`. The user can supply his/her own dense Jacobian function, or use the default difference quotient function `CVDenseDQJac` that comes with the CVDENSE solver. To specify a user-supplied Jacobian function `djac` and associated user data `jac_data`, CVDENSE provides the function `CVDenseSetJacFn`. The CVDENSE solver passes the pointer `jac_data` to its 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 `jac_data` may be identical to `f_data`, if the latter was specified through `CVodeSetFdata`.

`CVDenseSetJacFn`

Call `flag = CVDenseSetJacFn(cvode_mem, djac, jac_data);`

Description The function `CVDenseSetJacFn` specifies the dense Jacobian approximation function to be used and the pointer to user data.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODE memory block.
- `djac` (`CVDenseJacFn`) user-defined dense Jacobian approximation function.
- `jac_data` (`void *`) pointer to the user-defined data structure.

Return value The return value `flag` (of type `int`) is one of
- `CVDENSE_SUCCESS` The optional value has been successfully set.
- `CVDENSE_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVDENSE_LMEM_NULL` The CVDENSE linear solver has not been initialized.

Notes By default, CVDENSE uses the difference quotient function `CVDenseDQJac`. If `NULL` is passed to `djac`, this default function is used.

The function type `CVDenseJacFn` is described in §5.6.3.

Band Linear solver. The CVBAND solver needs a function to compute a banded approximation to the Jacobian matrix $J(t, y)$. This function must be of type `CVBandJacFn`. The user can supply his/her own banded Jacobian approximation function, or use the default difference quotient function `CVBandDQJac` that comes with the CVBAND solver. To specify a user-supplied Jacobian function `bjac` and associated user data `jac_data`, CVBAND provides the function `CVBandSetJacFn`. The CVBAND solver passes the pointer `jac_data` to its 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 `jac_data` may be identical to `f_data`, if the latter was specified through `CVodeSetFdata`.

`CVBandSetJacFn`

Call `flag = CVBandSetJacFn(cvode_mem, bjac, jac_data);`

Description The function `CVBandSetJacFn` specifies the banded Jacobian approximation function to be used and the pointer to user data.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODE memory block.
- `bjac` (`CVBandJacFn`) user-defined banded Jacobian approximation function.
- `jac_data` (`void *`) pointer to the user-defined data structure.

Return value The return value `flag` (of type `int`) is one of
- `CVBAND_SUCCESS` The optional value has been successfully set.
- `CVBAND_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVBAND_LMEM_NULL` The CVBAND linear solver has not been initialized.

Notes By default, CVBAND uses the difference quotient function `CVBandDQJac`. If `NULL` is passed to `bjac`, this default function is used.

The function type `CVBandJacFn` is described in §5.6.4.

SPGMR Linear solver. If any type of preconditioning is to be done within the SPGMR method, then the user must supply a preconditioner solve function `psolve` and specify its name through a call to `CVSpgmrSetPreconditioner`. 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 §5.6. If used, the `psetup` function should also be specified in the call to `CVSpgmrSetPreconditioner`. Optionally, the CVSPGMR solver passes the pointer it receives through `CVSpgmrSetPreconditioner` to the preconditioner setup and solve functions. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied preconditioner functions without using global data in the program. The pointer `p_data` may be identical to `f_data`, if the latter was specified through `CVodeSetFdata`.

The CVSPGMR solver requires a function to compute an approximation to the product between the Jacobian matrix $J(t, y)$ and a vector v. The user can supply his/her own Jacobian times vector approximation function, or use the difference quotient function `CVSpgmrDQJtimes` that comes with the CVSPGMR solver. A user-defined Jacobian-vector function must be of type `CVSpgmrJtimesFn` and can be specified through a call to `CVSpgmrSetJacTimesVecFn` (see §5.6 for specification details). As with the preconditioner user data structure `p_data`, the user can also specify int the call to `CVSpgmrSetJacFn`, a pointer to a user-defined data structure, `jac_data`, which the CVSPGMR solver passes to the Jacobian times vector function `jtimes` each time it is called. The pointer `jac_data` may be identical to `p_data` and/or `f_data`.

`CVSpgmrSetPreconditioner`

Call `flag = CVSpgmrSetPrecSolveFn(cvode_mem, psolve, psetup, p_data);`

Description The function `CVSpgmrSet` specifies the preconditioner setup and solve functions and the pointer to user data.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODE memory block.
- `psolve` (`CVSpgmrPrecSolveFn`) user-defined preconditioner solve function.
- `psetup` (`CVSpgmrPrecSetupFn`) user-defined preconditioner setup function.
- `p_data` (`void *`) pointer to the user-defined data structure.

Return value The return value `flag` (of type `int`) is one of
- `CVSPGMR_SUCCESS` The optional value has been successfully set.
- `CVSPGMR_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVSPGMR_LMEM_NULL` The CVSPGMR linear solver has not been initialized.

Notes The function type `CVSpgmrPrecSolveFn` is described in §5.6.6. The function type `CVSpgmrPrecSetupFn` is described in §5.6.7.

`CVSpgmrSetJacTimesVecFn`

Call `flag = CVSpgmrSetJacTimesVecFn(cvode_mem, jtimes, jac_data);`

Description The function `CVSpgmrSetJacTimesFn` specifies the Jacobian-vector function to be used and the pointer to user data.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODE memory block.
- `jtimes` (`CVSpgmrJacTimesVecFn`) user-defined Jacobian-vector product function.
- `jac_data` (`void *`) pointer to the user-defined data structure.

Return value The return value `flag` (of type `int`) is one of
- `CVSPGMR_SUCCESS` The optional value has been successfully set.
- `CVSPGMR_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVSPGMR_LMEM_NULL` The CVSPGMR linear solver has not been initialized.

Notes: By default, CVSPGMR uses the difference quotient function `CVSpgmrDQJtimes`. If `NULL` is passed to `jtimes`, this default function is used.

The function type `CVSpgmrJacTimesVecFn` is described in §5.6.5.

CVSpgmrSetGSType

Call: `flag = CVSpgmrSetGSType(cvode_mem, gstype);`

Description: The function `CVSpgmrSetGSType` specifies the type of Gram-Schmidt orthogonalization to be used. 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.
- `gstype` (`int`) type of Gram-Schmidt orthogonalization.

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

- `CVSPGMR_SUCCESS` The optional value has been successfully set.
- `CVSPGMR_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVSPGMR_LMEM_NULL` The CVSPGMR linear solver has not been initialized.
- `CVSPGMR_ILL_INPUT` The Gram-Schmidt orthogonalization type `gstype` is not valid.

Notes: The default value is `MODIFIED_GS`.

CVSpgmrSetDelt

Call: `flag = CVSpgmrSetDelt(cvode_mem, delt);`

Description: The function `CVSpgmrSetDelt` specifies the factor by which the GMRES convergence test constant is reduced from the Newton iteration test constant.

Arguments:
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `delt` (`realtype`) ratio between linear and nonlinear tolerances.

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

- `CVSPGMR_SUCCESS` The optional value has been successfully set.
- `CVSPGMR_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVSPGMR_LMEM_NULL` The CVSPGMR linear solver has not been initialized.
- `CVSPGMR_ILL_INPUT` The factor `delt` is negative.

Notes: The default value is 0.05.

Passing a value `delt = 0.0` also indicates the default value should be used.

CVSpgmrSetPrecType

Call: `flag = CVSpgmrSetPrecType(cvode_mem, pretype);`

Description: The function `CVSpgmrSetPrecType` 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: `PREC_NONE`, `PREC_LEFT`, `PREC_RIGHT`, or `PREC_BOTH`.

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

- `CVSPGMR_SUCCESS` The optional value has been successfully set.
- `CVSPGMR_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVSPGMR_LMEM_NULL` The CVSPGMR linear solver has not been initialized.
- `CVSPGMR_ILL_INPUT` The preconditioner type `pretype` is not valid.

Notes The preconditioning type is initially specified in the call to CVSpgmr (see §5.5.2). This function call is needed only if pretype is being changed from its value in the prior call to CVSpgmr.

5.5.5 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 $t_n - h_u \leq \mathtt{t} \leq t_n$, t_n denotes the current internal time reached, and h_u is the last internal step size successfully used by the solver. The user may request $\mathtt{k} = 0, 1, ..., q_u$, where q_u is the current order.

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 $0, 1, ..., q_u$.

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.

5.5.6 Optional output functions

CVODES provides an extensive set of functions that can be used to obtain solver performance information. Table 5.2 lists all optional output functions in CVODES, which are then described in detail in the remainder of this section.

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 5.2: Optional outputs from CVODES, CVDENSE, CVBAND, CVDIAG, and CVSPGMR

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 root function	CVodeGetNumGEvals
CVDENSE linear solver	
Size of CVDENSE real and integer workspaces	CVDenseGetWorkSpace
No. of Jacobian evaluations	CVDenseGetNumJacEvals
No. of r.h.s. calls for finite diff. Jacobian evals.	CVDenseGetNumRhsEvals
Last return from a CVDENSE function	CVDenseGetLastFlag
CVBAND linear solver	
Size of CVBAND real and integer workspaces	CVBandGetWorkSpace
No. of Jacobian evaluations	CVBandGetNumJacEvals
No. of r.h.s. calls for finite diff. Jacobian evals.	CVBandGetNumRhsEvals
Last return from a CVBAND function	CVBandGetLastFlag
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
CVSPGMR linear solver	
Size of CVSPGMR real and integer workspaces	CVSpgmrGetWorkSpace
No. of linear iterations	CVSpgmrGetNumLinIters
No. of linear convergence failures	CVSpgmrGetNumConvFails
No. of preconditioner evaluations	CVSpgmrGetNumPrecEvals
No. of preconditioner solves	CVSpgmrGetNumPrecSolves
No. of Jacobian-vector product evaluations	CVSpgmrGetNumJtimesEvals
No. of r.h.s. calls for finite diff. Jacobian-vector evals.	CVSpgmrGetNumRhsEvals
Last return from a CVSPGMR function	CVSpgmrGetLastFlag

`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 and maximum method order `maxord`, the actual size of the real workspace given in `realtype` words is:

- Base value: `lenrw` $= 86+$ (`maxord+5`)N
- With `itol = CV_SV`: `lenrw` $=$ `lenrw` $+N$
- With rootfinding for N_g functions (see §5.8): `lenrw` $=$ `lenrw` $+3N_g$

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

- Base value: `leniw` $= 52+$ (`maxord+5`)N
- With `itol = CV_SV`: `leniw` $=$ `leniw` $+N$
- With rootfinding for N_g functions: `leniw` $=$ `leniw` $+N_g$

For the default value of `maxord`, the base values are:

- For the Adams method: `lenrw` $= 96 + 17N$ and `leniw` $= 52 + 17N$
- For the BDF method: `lenrw` $= 96 + 10N$ and `leniw` $= 52 + 10N$

Note that additional memory is allocated if quadratures and/or forward sensitivity integration is enabled. See §5.7.1 and §6.2.1 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 ($h_{\min} \leq h_0 \leq h_{\max}$), 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 §3.4).

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`.
`CV_NO_SLDET` The stability limit detection algorithm was not activated through a call to `CVodeSetStabLimDet`.

`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 (3.7).

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`.

`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`.

Linear solver optional output functions

For each of the linear system solver modules there are various optional outputs that describe the performance of the module. The functions available to access these are described below.

Dense Linear solver. The following optional outputs are available from the CVDENSE module: 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 CVDENSE function.

`CVDenseGetWorkSpace`

Call `flag = CVDenseGetWorkSpace(cvode_mem, &lenrwD, &leniwD);`

Description The function `CVDenseGetWorkSpace` returns the CVDENSE real and integer workspace sizes.

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

Return value The return value `flag` (of type `int`) is one of
- `CVDENSE_SUCCESS` The optional output values have been successfully set.
- `CVDENSE_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVDENSE_LMEM_NULL` The CVDENSE linear solver has not been initialized.

Notes 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.

`CVDenseGetNumJacEvals`

Call `flag = CVDenseGetNumJacEvals(cvode_mem, &njevalsD);`

Description The function `CVDenseGetNumJacEvals` returns the number of calls made to the dense Jacobian approximation function.

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

Return value The return value `flag` (of type `int`) is one of
- `CVDENSE_SUCCESS` The optional output value has been successfully set.
- `CVDENSE_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVDENSE_LMEM_NULL` The CVDENSE linear solver has not been initialized.

`CVDenseGetNumRhsEvals`

Call `flag = CVDenseGetNumRhsEvals(cvode_mem, &nfevalsD);`

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `nfevalsD` (`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
- `CVDENSE_SUCCESS` The optional output value has been successfully set.
- `CVDENSE_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVDENSE_LMEM_NULL` The CVDENSE linear solver has not been initialized.

Notes The value `nfevalsD` is incremented only if the default `CVDenseDQJac` difference quotient function is used.

`CVDenseGetLastFlag`

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

Description The function `CVDenseGetLastFlag` returns the last return value from a CVDENSE routine.

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

Return value The return value `flag` (of type `int`) is one of
- `CVDENSE_SUCCESS` The optional output value has been successfully set.
- `CVDENSE_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVDENSE_LMEM_NULL` The CVDENSE linear solver has not been initialized.

Notes If the CVDENSE setup function failed (`CVode` returned `CV_LSETUP_FAIL`), then the value of `lsflag` corresponds to the column index (numbered from one) of a diagonal element with value zero that was encountered during the LU factorization of the dense Jacobian matrix.

Band Linear solver. The following optional outputs are available from the CVBAND module: 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 CVBAND function.

`CVBandGetWorkSpace`

Call `flag = CVBandGetWorkSpace(cvode_mem, &lenrwB, &leniwB);`

Description The function `CVBandGetWorkSpace` returns the CVBAND real and integer workspace sizes.

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

Return value The return value `flag` (of type `int`) is one of
- `CVBAND_SUCCESS` The optional output values have been successfully set.
- `CVBAND_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVBAND_LMEM_NULL` The CVBAND linear solver has not been initialized.

Notes — In terms of the problem size N and Jacobian half-bandwidths, the actual size of the real workspace is (2 `mupper`+3 `mlower`+2) N `realtype` words, and the actual size of the integer workspace is N integer words.

`CVBandGetNumJacEvals`

Call — `flag = CVBandGetNumJacEvals(cvode_mem, &njevalsB);`

Description — The function `CVBandGetNumJacEvals` returns the number of calls made to the banded Jacobian approximation function.

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

Return value — The return value `flag` (of type `int`) is one of
- `CVBAND_SUCCESS` The optional output value has been successfully set.
- `CVBAND_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVBAND_LMEM_NULL` The CVBAND linear solver has not been initialized.

`CVBandGetNumRhsEvals`

Call — `flag = CVBandGetNumRhsEvals(cvode_mem, &nfevalsB);`

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `nfevalsB` (`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
- `CVBAND_SUCCESS` The optional output value has been successfully set.
- `CVBAND_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVBAND_LMEM_NULL` The CVBAND linear solver has not been initialized.

Notes — The value `nfevalsB` is incremented only if the default `CVBandDQJac` difference quotient function is used.

`CVBandGetLastFlag`

Call — `flag = CVBandGetLastFlag(cvode_mem, &lsflag);`

Description — The function `CVBandGetLastFlag` returns the value of the last return flag from a CVBAND routine.

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

Return value — The return value `flag` (of type `int`) is one of
- `CVBAND_SUCCESS` The optional output value has been successfully set.
- `CVBAND_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVBAND_LMEM_NULL` The CVBAND linear solver has not been initialized.

Notes — If the CVBAND setup function failed (`CVode` returned `CV_LSETUP_FAIL`), the value of `lsflag` corresponds to the column index (numbered from one) of a diagonal element with value zero that was encountered during the LU factorization of the banded Jacobian matrix.

Diagonal Linear solver. 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.

`CVDiagGetWorkSpace`

Call `flag = CVDiagGetWorkSpace(cvode_mem, &lenrwDI, &leniwDI);`

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `lenrwDI` (`long int`) the number of `realtype` values in the CVDIAG workspace.
- `leniwDI` (`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 $3N$ `realtype` words.

`CVDiagGetNumRhsEvals`

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

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.
- `nfevalsDI` (`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 (available by calling `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` (`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`).

SPGMR Linear solver. The following optional outputs are available from the CVSPGMR module: 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 CVSPGMR function.

`CVSpgmrGetWorkSpace`

Call `flag = CVSpgmrGetWorkSpace(cvode_mem, &lenrwSG, &leniwSG);`

Description The function `CVSpgmrGetWorkSpace` returns the CVSPGMR real and integer workspace sizes.

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

Return value The return value `flag` (of type `int`) is one of
- `CVSPGMR_SUCCESS` The optional output values have been successfully set.
- `CVSPGMR_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVSPGMR_LMEM_NULL` The CVSPGMR 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 (`maxl`+5) $*$ N+ `maxl` $*$(`maxl`+4) + 1 `realtype` words. (In a parallel setting, this value is global - summed over all processes.)

`CVSpgmrGetNumLinIters`

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

Description The function `CVSpgmrGetNumLinIters` 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
- `CVSPGMR_SUCCESS` The optional output value has been successfully set.
- `CVSPGMR_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVSPGMR_LMEM_NULL` The CVSPGMR linear solver has not been initialized.

`CVSpgmrGetNumConvFails`

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

Description The function `CVSpgmrGetNumConvFails` 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
- `CVSPGMR_SUCCESS` The optional output value has been successfully set.
- `CVSPGMR_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CVSPGMR_LMEM_NULL` The CVSPGMR linear solver has not been initialized.

`CVSpgmrGetNumPrecEvals`

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

Description The function `CVSpgmrGetNumPrecEvals` 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

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

`CVSPGMR_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CVSPGMR_LMEM_NULL` The CVSPGMR linear solver has not been initialized.

`CVSpgmrGetNumPrecSolves`

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

Description The function `CVSpgmrGetNumPrecSolves` 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

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

`CVSPGMR_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CVSPGMR_LMEM_NULL` The CVSPGMR linear solver has not been initialized.

`CVSpgmrGetNumJtimesEvals`

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

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

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

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

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

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

`CVSPGMR_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CVSPGMR_LMEM_NULL` The CVSPGMR linear solver has not been initialized.

`CVSpgmrGetNumRhsEvals`

Call `flag = CVSpgmrGetNumRhsEvals(cvode_mem, &nfevalsSG);`

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

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

`nfevalsSG` (`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

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

`CVSPGMR_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CVSPGMR_LMEM_NULL` The CVSPGMR linear solver has not been initialized.

Notes The value `nfevalsSG` is incremented only if the default `CVSpgmrDQJtimes` difference quotient function is used.

`CVSpgmrGetLastFlag`

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

Description The function `CVSpgmrGetLastFlag` returns the last return value from a CVSPGMR routine.

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

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

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

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

`CVSPGMR_MEM_NULL` The `cvode_mem` pointer is `NULL`.

`CVSPGMR_LMEM_NULL` The CVSPGMR linear solver has not been initialized.

Notes If the CVSPGMR setup function failed (`CVode` returned `CV_LSETUP_FAIL`), then `lsflag` contains the return value of the preconditioner setup function `psetup`.

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_CONV_FAIL` indicating a failure to converge, `SPGMR_QRFACT_FAIL` indicating a singular matrix was found during the QR factorization, `SPGMR_PSOLVE_FAIL_REC` indicating that the preconditioner solve function `psolve` failed recoverably, `SPGMR_MEM_NULL` indicating that the SPGMR memory is `NULL`, `SPGMR_ATIMES_FAIL` indicating a failure of the Jacobian-vector product 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.

5.5.7 CVODES reinitialization function

The function `CVodeReInit` reinitializes the main CVODES solver for the solution of a problem, where a prior call to `CVodeMalloc` has been made. The new problem must have the same size as the previous one. `CVodeReInit` performs the same input checking and initializations that `CVodeMalloc` does, but does no memory allocation as it assumes that the existing internal memory is sufficient for the new problem.

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

If there are changes to the linear solver specifications, make the appropriate `CV*Set*` calls, as described in §5.5.2

`CVodeReInit`

Call `flag = CVodeReInit(cvode_mem, f, t0, y0, itol, reltol, abstol);`

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

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

`f` (`CVRhsFn`) is the C function which computes f in the ODE. This function has the form `f(N, t, y, ydot, f_data)` (for full details see §5.6).

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

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

itol	(int) is one of CV_SS, CV_SV, or CV_WF, where itol = CV_SS indicates scalar relative error tolerance and scalar absolute error tolerance, while itol = CV_SV indicates scalar relative error tolerance and vector absolute error tolerance. The latter choice is important when the absolute error tolerance needs to be different for each component of the ODE. If itol=CV_WF, the arguments reltol and abstol are ignored and the user is expected to provide a function to evaluate the error weight vector W from (3.7). See CVodeSetEwtFn in §5.5.4.
reltol	(realtype) is the relative error tolerance.
abstol	(void *) is a pointer to the absolute error tolerance. If itol=CV_SS, abstol must be a pointer to a realtype variable. If itol=CV_SV, abstol must be an N_Vector variable.

Return value The return flag 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 CVodeMalloc.
CV_ILL_INPUT	An input argument to CVodeReInit has an illegal value.

Notes If an error occurred, CVodeReInit also prints an error message to the file specified by the optional input errfp.

! It is the user's responsibility to provide compatible itol and abstol arguments.

5.6 User-supplied functions

The user-supplied functions consist of one function defining the ODE, (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 the SPGMR algorithm.

5.6.1 ODE right-hand side

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

CVRhsFn

Definition
```
typedef void (*CVRhsFn)(realtype t, N_Vector y, N_Vector ydot,
                        void *f_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 $f(t, y)$.
f_data	is a pointer to user data — the same as the f_data parameter passed to CVodeSetFdata.

Return value A CVRhsFn function type does not have a return value.

Notes Allocation of memory for ydot is handled within CVODES.

5.6.2 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 $\| v\|_{\mathrm{WRMS}} = \sqrt{(1/N)\sum_1^N W_i \cdot v_i}$. The function type `CVEwtFn` is defined as follows:

CVEwtFn

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

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

Arguments
- `y` is the value of the vector for which the WRMS norm must be computed.
- `ewt` is the output vector containing the error weights.
- `e_data` is a pointer to user data — the same as the `e_data` parameter passed to `CVodeSetEwtFn`.

Return value A `CVEwtFn` function type must return 0 if it successfuly set the error weights and -1 otherwise. In case of failure, a message is printed and the integration stops.

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.

5.6.3 Jacobian information (direct method with dense Jacobian)

If the direct linear solver with dense treatment of the Jacobian is used (i.e., `CVDense` is called in Step 7 of §5.4), the user may provide a function of type `CVDenseJacFn` defined by:

CVDenseJacFn

Definition
```
typedef void (*CVDenseJacFn)(long int N, DenseMat J, realtype t,
                             N_Vector y, N_Vector fy, void *jac_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.
- `J` is the output Jacobian matrix.
- `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 vector $f(t, y)$.
- `jac_data` is a pointer to user data — the same as the `jac_data` parameter passed to `CVDenseSetJacData`.
- `tmp1`
- `tmp2`
- `tmp3` are pointers to memory allocated for variables of type `N_Vector` which can be used by `CVDenseJacFn` as temporary storage or work space.

Return value A `CVDenseJacFn` function type does not have a return value.

Notes A user-supplied dense Jacobian function must load the `N` by `N` dense matrix `J` with an approximation to the Jacobian matrix J at the point (`t`, `y`). Only nonzero elements need to be loaded into `J` because `J` is set to the zero matrix before the call to the Jacobian function. The type of `J` is `DenseMat`.

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 DenseMat type. DENSE_ELEM(J, i, j) references the (i, j)-th element of the dense matrix J (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 J (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 DenseMat type and accessor macros DENSE_ELEM and DENSE_COL are documented in §10.1.

If the user's CVDenseJacFn 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, use the CVodeGet* functions described in §5.5.6. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundialstypes.h.

5.6.4 Jacobian information (direct method with banded Jacobian)

If the direct linear solver with banded treatment of the Jacobian is used (i.e. CVBand is called in Step 7 of §5.4), the user may provide a function of type CVBandJacFn defined as follows:

CVBandJacFn

Definition
```
typedef void (*CVBandJacFn)(long int N, long int mupper,
                            long int mlower, BandMat J, realtype t,
                            N_Vector y, N_Vector fy, void *jac_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.
- J is the output Jacobian matrix.
- 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 vector $f(t, y)$.
- jac_data is a pointer to user data - the same as the jac_data parameter passed to CVBandSetJacData.
- tmp1
- tmp2
- tmp3 are pointers to memory allocated for variables of type N_Vector which can be used by CVBandJacFn as temporary storage or work space.

Return value: A CVBandJacFn function type does not have a return value.

Notes: A user-supplied band Jacobian function must load the band matrix J of type BandMat with the elements of the Jacobian $J(t, y)$ at the point (t,y). Only nonzero elements need to be loaded into J because J 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 BandMat type. BAND_ELEM(J, i, j) references the (i, j)-th element of the band matrix J, 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 J, 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 BandMat. 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 BandMat type and the accessor macros BAND_ELEM, BAND_COL and BAND_COL_ELEM are documented in §10.2.

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, use the CVodeGet* functions described in §5.5.6. The unit roundoff can be accessed as UNIT_ROUNDOFF defined in sundialstypes.h.

5.6.5 Jacobian information (SPGMR matrix-vector product)

If an iterative SPGMR linear solver is selected (CVSpgmr is called in step 7 of §5.4), the user may provide a function of type CVSpgmrJacTimesVecFn defined as follows:

CVSpgmrJacTimesVecFn

Definition

```
typedef int (*CVSpgmrJacTimesVecFn)(N_Vector v, N_Vector Jv, realtype t,
                                    N_Vector y, N_Vector fy,
                                    void *jac_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 vector $f(t, y)$.
- jac_data is a pointer to user data - the same as the jac_data parameter passed to CVSpgmrSetJacData.
- 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 SPGMR generic solver, in which case the integration is halted.

Notes If the user's CVSpgmrJacTimesVecFn 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, use the `CVodeGet*` functions described in §5.5.6. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundialstypes.h`.

5.6.6 Preconditioning (SPGMR 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. This function must be of type `CVSpgmrPrecSolveFn`, defined as follows:

`CVSpgmrPrecSolveFn`

Definition
```
typedef int (*CVSpgmrPrecSolveFn)(realtype t, N_Vector y, N_Vector fy,
                                  N_Vector r, N_Vector z,
                                  realtype gamma, realtype delta,
                                  int lr, void *p_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 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 §5.5.6).
- `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`);
- `p_data` is a pointer to user data - the same as the `p_data` parameter passed to the function `CVSpgmrSetPrecData`.
- `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).

5.6.7 Preconditioning (SPGMR 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 `CVSpgmrPrecSetupFn`, defined as follows:

`CVSpgmrPrecSetupFn`

Definition
```
typedef int (*CVSpgmrPrecSetupFn)(realtype t, N_Vector y, N_Vector fy,
                                  booleantype jok, booleantype *jcurPtr,
                                  realtype gamma, void *p_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 `CVSpgmrPrecSetupFn` 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 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 $M = I - \gamma P$.
- `p_data` is a pointer to user data, the same as the `p_data` parameter passed to `CVSpgmrSetPrecData`.
- `tmp1`
- `tmp2`
- `tmp3` are pointers to memory allocated for variables of type `N_Vector` which can be used by `CVSpgmrPrecSetupFn` 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 `CVSpgmrPrecSetupFn` 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, use the `CVodeGet*` functions described in §5.5.6. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundialstypes.h`.

5.7 Integration of pure quadrature equations

If the system of ODEs contains *pure quadratures*, it is more efficient to treat them separately by excluding them from the nonlinear solution stage. To do this, begin by excluding the quadrature variables from the vector `y` and the quadrature equations from within `f`. 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 §5.4 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 `CVodeQuadMalloc` to specify the quadrature equation right-hand side function and to allocate internal memory related to quadrature integration. See §5.7.1 for details.
11. **Set optional inputs for quadrature integration**

    Call `CVodeSetQuadFdata` to specify user data required for the evaluation of the quadrature equation right-hand side. 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 §5.7.3 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 §5.7.2 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 §5.7.4 for details.
16. **Deallocate memory for solution vector and for the vector of quadrature variables**
17. **Free solver memory**
18. **[P] Finalize MPI**

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

5.7.1 Quadrature initialization functions

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

`CVodeQuadMalloc`

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

Description The function `CVodeQuadMalloc` 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 §5.7.5).
- `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 `CVodeQuadMalloc` was successful.
- `CV_MEM_NULL` The CVODES memory block was not initialized by a prior call to `CVodeCreate`.
- `CV_MEM_FAIL` A memory allocation request failed.

Notes If an error occured, `CVodeQuadMalloc` also prints an error message to the file specified by the optional input `errfp`.

If quadrature integration is enabled, the

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

- Base value: `lenrw` = `lenrw` + (`maxord+5`)N_q
- With `itolQ = CV_SV` (see `CVodeSetQuadErrCon`): `lenrw` = `lenrw` +N_q

the size of the integer workspace is increased by:

- Base value: `leniw` = `leniw` + (`maxord+5`)N_q
- With `itolQ = CV_SV`: `leniw` = `leniw` +N_q

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

`CVodeQuadReInit`

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

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `fQ` (`CVQuadRhsFn`) is the C function which computes fQ, the right-hand side of the quadrature equations.
- `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 `CVodeReInit` was successful.
- `CV_MEM_NULL` The CVODES memory block was not initialized by a prior call to `CVodeCreate`.

CV_NO_QUAD	Memory space for the quadrature integration was not allocated by a prior call to CVodeQuadMalloc.

Notes If an error occured, CVodeQuadReInit also prints an error message to the file specified by the optional input errfp.

5.7.2 Quadrature extraction functions

If quadrature integration has been initialized by a call to CVodeQuadMalloc, 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 y. Solution quadratures can be obtained using the following function:

CVodeGetQuad

Call `flag = CVodeGetQuad(cvode_mem, t, 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 CVodeMalloc.
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.
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.
CV_BAD_T	The time t is not in the allowed range.

Notes In case of an error return, an error message is also printed.

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, 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 CVodeMalloc.
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.
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 $0, 1, ..., q_u$.
CV_BAD_T	The time t is not in the allowed range.

Notes: In case of an error return, an error message is also printed.

5.7.3 Optional inputs for quadrature integration

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

CVodeSetQuadFdata

Call: `flag = CVodeSetQuadFdata(cvode_mem, fQ_data);`

Description: The function `CVodeSetQuadFdata` specifies the user-defined data block `fQ_data` and attaches it to the main CVODES memory block.

Arguments:
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `fQ_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 `fQ_data` is not specified, a `NULL` pointer is passed to all user-supplied functions that have it as an argument. Note that `fQ_data` can be the same as the pointer `f_data` set through `CVodeSetFdata`.

CVodeSetQuadErrCon

Call: `flag = CVodeSetQuadErrCon(cvode_mem, errconQ, itolQ, reltolQ, abstolQ);`

Description: The function `CVodeSetQuadErrCon` specifies whether or not quadrature variables shoule be used in the step size control mechanism, and if so, specifies the integration tolerances for 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. If `errconQ=FALSE`, the following three arguments are ignored.
- `itolQ` (`int`) is either `CV_SS` or `CV_SV`, where `itolQ = CV_SS` indicates scalar relative error tolerance and scalar absolute error tolerance, while `itolQ = CV_SV` indicates scalar relative error tolerance and vector absolute error tolerance. The latter choice is important when the absolute error tolerance needs to be different for each quadrature variable.
- `reltolQ` (`realtype *`) is a pointer to the relative error tolerance.
- `abstolQ` (`void *`) is a pointer to the absolute error tolerance. If `itolQ=CV_SS`, `abstolQ` must be a pointer to a `realtype` variable. If `itolQ=CV_SV`, `abstolQ` must be an `N_Vector` variable.

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 input argument to `CVodeSetQuadErrCon` has an illegal value.

Notes: By default, `errconQ` is set to `FALSE`.

! It is illegal to call `CVodeSetQuadErrCon` before a call to `CVodeQuadMalloc`.

5.7.4 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.

5.7.5 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. This function must be of type `CVQuadRhsFn` defined as follows:

`CVQuadRhsFn`

Definition
```
typedef void (*CVQuadRhsFn)(realtype t, N_Vector y,
                            N_Vector yQdot, void *fQ_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 $fQ(t, y)$.
- `fQ_data` is a pointer to user data - the same as the `fQ_data` parameter passed to `CVodeSetQuadFdata`.

Return value A `CVQuadRhsFn` function type does not have a return value.

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 §8.1 and §8.2).

5.8 Rootfinding

While solving the IVP, CVODES has the capability to find the roots of a set of user-defined functions. This section describes the user-callable functions used to initialize and define the rootfinding problem and to obtain solution information, and it also describes the required user-supplied function.

5.8.1 User-callable functions for rootfinding

`CVodeRootInit`

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

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

Arguments
- `cvode_mem` (`void *`) pointer to the CVODE memory block returned by `CVodeCreate`.
- `nrtfn` (`int`) is the number of root functions g_i.
- `g` (`CVRootFn`) is the C function which defines the `nrtfn` functions $g_i(t, y)$ whose roots are sought. See §5.8.2 for details.
- `g_data` (`void *`) pointer to the user data for use by the user's root function g.

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_RTFUNC_NULL` 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`$= 0$.

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 *`) an `int` array of length `nrtfn` showing the indices of the user functions g_i found to have a root. For $i = 0, \ldots,$`nrtfn`-1, `rootsfound`$[i]=$ 1 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 ! 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`.

5.8.2 User-supplied function for rootfinding

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 void (*CVRootFn)(realtype t, N_Vector y, realtype *gout,
                          void *g_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 $g_i(t, y)$.
- `g_data` is a pointer to user data - the same as the `g_data` parameter passed to `CVodeSetGdata`.

Return value
A `CVRootFn` function type does not have a return value.

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

5.9 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.

5.9.1 A serial banded preconditioner module

This preconditioner provides a band matrix preconditioner based on difference quotients of the ODE right-hand side function `f`. It generates 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 in CVSPGMR. 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 §5.3), to use the CVBANDPRE module, the main program must include the header file `cvbandpre.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 §5.4 are grayed out.

1. **Set problem dimensions**
2. **Set vector of initial values**
3. **Create CVODES object**
4. **Set optional inputs**
5. **Allocate internal memory**
6. **Initialize the CVBANDPRE preconditioner module**

   Specify the upper and lower half-bandwidths (`mu` and `ml`, respectively) and call

   `bp_data = CVBandPrecAlloc(cvode_mem, N, mu, ml);`

   to allocate memory for and to initialize a data structure (pointed to by `bp_data`) to be passed to the CVSPGMR linear solver.

7. **Attach the CVSPGMR linear solver**

   `flag = CVBPSpgmr(cvode_mem, pretype, maxl, bp_data);`

   The function `CVBPSpgmr` is a wrapper for the CVSPGMR specification function `CVSpgmr` and performs the following actions:

   - Attaches the CVSPGMR linear solver to the main CVODES solver memory;
   - Sets the preconditioner data structure for CVBANDPRE;
   - Sets the preconditioner setup function for CVBANDPRE;
   - Sets the preconditioner solve function for CVBANDPRE;

   The arguments `pretype` and `maxl` are described below. The last argument to `CVBPSpgmr` is a pointer to the CVBANDPRE data structure returned by `CVBandPrecAlloc`.

8. **Set linear solver optional inputs**

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

9. **Advance solution in time**

10. **Deallocate memory for solution vector**

11. **Free the CVBANDPRE data structure**

    `CVBandPrecFree(bp_data);`

12. **Free solver memory**

The three user-callable functions that initialize, attach, and deallocate the CVBANDPRE preconditioner module (steps 6, 7 and 11 above) are described in more detail below.

`CVBandPrecAlloc`

Call	`bp_data = CVBandPrecAlloc(cvode_mem, N, mu, ml);`
Description	The function `CVBandPrecAlloc` initializes and allocates memory for the CVBANDPRE preconditioner.
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	If successful, `CVBandPrecAlloc` returns a pointer to the newly created CVBANDPRE memory block (of type `void *`). If an error occurred, `CVBandPrecAlloc` returns `NULL`.
Notes	The banded approximate Jacobian will have nonzero elements only in locations (i, j) with $-\mathtt{ml} \leq j - i \leq \mathtt{mu}$.

`CVBPSpgmr`

Call	`flag = CVBPSpgmr(cvode_mem, pretype, maxl, bp_data);`
Description	The function `CVBPSpgmr` links the CVBANDPRE data to the CVSPGMR linear solver and attaches the latter to the CVODES memory block.
Arguments	`cvode_mem` (`void *`) pointer to the CVODES memory block. `pretype` (`int`) specifies the preconditioning type and must be either `PREC_LEFT` or `PREC_RIGHT`.

`maxl`	(`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `CVSPGMR_MAXL = 5`.
`bp_data`	(`void *`) pointer to the CVBANDPRE data structure.

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

`CVSPGMR_SUCCESS`	The CVSPGMR initialization was successful.
`CVSPGMR_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CVSPGMR_ILL_INPUT`	The preconditioner type `pretype` is not valid.
`CVSPGMR_MEM_FAIL`	A memory allocation request failed.
`CV_PDATA_NULL`	The CVBANDPRE preconditioner has not been initialized.

`CVBandPrecFree`

Call `CVBandPrecFree(bp_data);`

Description The function `CVBandPrecFree` frees the pointer allocated by `CVBandPrecAlloc`.

Arguments The only argument passed to `CVBandPrecFree` is the pointer to the CVBANDPRE data structure (of type `void *`).

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

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

`CVBandPrecGetWorkSpace`

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

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

Arguments

`bp_data`	(`void *`) pointer to the CVBANDPRE data structure.
`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:

`CV_SUCCESS`	The optional output values have been successfully set.
`CV_PDATA_NULL`	The CVBANDPRE preconditioner has not been initialized.

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

`CVBandPrecGetNumRhsEvals`

Call `flag = CVBandPrecGetNumRhsEvals(bp_data, &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

`bp_data`	(`void *`) pointer to the CVBANDPRE data structure.
`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:

`CV_SUCCESS`	The optional output value has been successfully set.
`CV_PDATA_NULL`	The CVBANDPRE preconditioner has not been initialized.

5.9.2 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 (3.5) 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 [19] and is included in a software module within the CVODES package. This module works with NVECTOR_PARALLEL and 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 (3.1). 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

$$g(t, y) = [g_1(t, \bar{y}_1), g_2(t, \bar{y}_2), \ldots, g_M(t, \bar{y}_M)]^T \tag{5.1}$$

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

$$P = diag[P_1, P_2, \ldots, P_M] \tag{5.2}$$

where

$$P_m \approx I - \gamma J_m \tag{5.3}$$

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 `mu` + `ml` +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

$$Px = b \tag{5.4}$$

reduces to solving each of the equations

$$P_m x_m = b_m \tag{5.5}$$

and this is done by banded LU factorization of P_m followed by a banded backsolve.

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 `f_data` that is passed by the user to `CVodeSetFdata` and that was passed to the user's function `f`, and neither function has a return value. The user is responsible for providing space (presumably within `f_data`) for components of `y` that are communicated between processes by `cfn`, and that are then used by `gloc`, which is not expected to do any communication.

`CVLocalFn`

Definition
```
typedef void (*CVLocalFn)(long int Nlocal, realtype t, N_Vector y,
                          N_Vector glocal, void *f_data);
```

Purpose This 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.
- `f_data` is a pointer to user data - the same as the `f_data` parameter passed to `CVodeSetFdata`.

Return value A `CVLocalFn` function type does not have a return value.

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

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

`CVCommFn`

Definition
```
typedef void (*CVCommFn)(long int Nlocal, realtype t,
                         N_Vector y, void *f_data);
```

Purpose This 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.
- `f_data` is a pointer to user data - the same as the `f_data` parameter passed to `CVodeSetFdata`.

Return value A `CVCommFn` function type does not have a return value.

Notes The `cfn` function is expected to save communicated data in space defined within the data structure `f_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 comunication was done in `f`, then `cfn = NULL` can be passed in the call to `CVBBDPrecAlloc` (see below).

Besides the header files required for the integration of the ODE problem (see §5.3), to use the CVBBDPRE module, the main program must include the header file `cvbbdpre.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 §5.4 are grayed out.

1. **Initialize MPI**
2. **Set problem dimensions**
3. **Set vector of initial values**
4. **Create CVODES object**

5. **Set optional inputs**
6. **Allocate internal memory**
7. **Initialize the CVBBDPRE preconditioner module**

   Specify the upper and lower half-bandwidths mudq and mldq, and mukeep and mlkeep, and call

   ```
   bbd_data = CVBBDPrecAlloc(cvode_mem, local_N, mudq, mldq,
                              mukeep, mlkeep, dqrely, gloc, cfn);
   ```

   to allocate memory for and to initialize a data structure (pointed to by bbd_data) to be passed to the CVSPGMR linear solver. The last two arguments passed to CVBBDPrecAlloc are the two mandatory user-supplied functions described above.

8. **Attach the CVSPGMR linear solver**

   ```
   flag = CVBBDSpgmr(cvode_mem, pretype, maxl, bbd_data);
   ```

   The function CVBPSpgmr is a wrapper for the CVSPGMR specification function CVSpgmr and performs the following actions:

   - Attaches the CVSPGMR linear solver to the main CVODES solver memory;
   - Sets the preconditioner data structure for CVBBDPRE;
   - Sets the preconditioner setup function for CVBBDPRE;
   - Sets the preconditioner solve function for CVBBDPRE;

   The arguments pretype and maxl are described below. The last argument passed to CVBBDSpgmr is a pointer to the CVBBDPRE memory block returned by CVBBDPrecAlloc.

9. **Set linear solver optional inputs**

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

10. **Advance solution in time**
11. **Deallocate memory for solution vector**
12. **Free the CVBBDPRE data structure**

    ```
    CVBBDPrecFree(bbd_data);
    ```

13. **Free solver memory**
14. **Finalize MPI**

The three user-callable functions that initialize, attach, and deallocate the CVBBDPRE preconditioner module (steps 7, 8, and 12 above) are described next.

CVBBDPrecAlloc

Call
```
bbd_data = CVBBDPrecAlloc(cvode_mem, local_N, mudq, mldq,
                           mukeep, mlkeep, dqrely, gloc, cfn);
```

Description The function CVBBDPrecAlloc initializes and allocates 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 If successful, `CVBBDPrecAlloc` returns a pointer to the newly created CVBBDPRE memory block (of type `void *`). If an error occurred, `CVBBDPrecAlloc` returns `NULL`.

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 with 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 for every process.

`CVBBDSpgmr`

Call `flag = CVBBDSpgmr(cvode_mem, pretype, maxl, bbd_data);`

Description The function `CVBBDSpgmr` links the CVBBDPRE data to the CVSPGMR linear solver and attaches the latter to the CVODES memory block.

Arguments

`cvode_mem`	(`void *`) pointer to the CVODES memory block.
`pretype`	(`int`) preconditioning type. Must be either `PREC_LEFT` or `PREC_RIGHT`.
`maxl`	(`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `CVSPGMR_MAXL = 5`.
`bbd_data`	(`void *`) pointer to the CVBBDPRE data structure.

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

`CVSPGMR_SUCCESS`	The CVSPGMR initialization was successful.
`CVSPGMR_MEM_NULL`	The `cvode_mem` pointer is `NULL`.
`CVSPGMR_ILL_INPUT`	The preconditioner type `pretype` is not valid.
`CVSPGMR_MEM_FAIL`	A memory allocation request failed.
`CV_PDATA_NULL`	The CVBBDPRE preconditioner has not been initialized.

`CVBBDPrecFree`

Call `CVBBDPrecFree(bbd_data);`

Description The function `CVBBDPrecFree` frees the pointer allocated by `CVBBDPrecAlloc`.

Arguments The only argument passed to `CVBBDPrecFree` is the pointer to the CVBBDPRE data structure (of type `void *`).

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

The CVBBDPRE module also provides a reinitialization function to allow solving a sequence of problems of the same size with CVSPGMR/CVBBDPRE, 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`.

`CVBBDPrecReInit`

Call `flag = CVBBDPrecReInit(bbd_data, mudq, mldq, dqrely, gloc, cfn);`

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

Arguments
- `bbd_data` (`void *`) pointer to the CVBBDPRE data structure.
- `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`.
- `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 of `CVBBDPrecReInit` is always `CV_SUCCESS`.

Notes If one of the half-bandwidths `mudq` or `mldq` is negative or exceeds the value `local_N`-1, it is replaced with 0 or `local_N`-1 accordingly.

The following two optional output functions are available for use with the CVBBDPRE module:

`CVBBDPrecGetWorkSpace`

Call `flag = CVBBDPrecGetWorkSpace(bbd_data, &lenrwBBDP, &leniwBBDP);`

Description The function `CVBBDPrecGetWorkSpace` returns the local CVBBDPRE real and integer workspace sizes.

Arguments
- `bbd_data` (`void *`) pointer to the CVBBDPRE data structure.
- `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:
- `CV_SUCCESS` The optional output values have been successfully set.
- `CV_PDATA_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.

`CVBBDPrecGetNumGfnEvals`

Call `flag = CVBBDPrecGetNumGfnEvals(bbd_data, &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
- `bbd_data` (`void *`) pointer to the CVBBDPRE data structure.
- `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
- `CV_SUCCESS` The optional output value has been successfully set.
- `CV_PDATA_NULL` The CVBBDPRE preconditioner has not been initialized.

The costs associated with CVBBDPRE also include `nlinsetups` LU factorizations, `nlinsetups` calls to `cfn`, and `npsolves` banded backsolve calls, where `nlinsetups` and `npsolves` are optional CVODES outputs (see §5.5.6).

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.

Chapter 6

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 (3.9). The only departure from this philosophy is due to the `CVRhsFn` type definition (§5.6). 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 Chapter 11.

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 §5.

6.1 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 §5.4, 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 with [**P**] correspond to NVECTOR_PARALLEL, while steps marked with [**S**] correspond to NVECTOR_SERIAL. Differences between the user main program in §5.4 and the one below start only at step (9).

First, note that no additional header files need be included for forward sensitivity analysis beyond those for IVP solution (§5.4).

1. **[P] Initialize MPI**

2. **Set problem dimensions**

3. **Set 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. **Define the sensitivity problem**

   - Number of sensitivities

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

   - Problem parameters

     If CVODES will 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 `f_data`. For example, `f_data->p = p;`

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

   - Parameter list

     If CVODES will evaluate the right-hand sides of the sensitivity systems, set `plist`, an array of `Ns` integer flags to specify the parameters `p` with respect to which solution sensitivities are to be computed. If sensitivities with respect to the j-th problem parameter are desired, set $\text{plist}_i = j$, for some $i = 1, \ldots, N_s$. Note that the parameters are numbered from 1 for the purpose of `plist`.

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

   - Parameter scaling factors

     If CVODES estimates tolerances for the sensitivity solution vectors (based on tolerances for the state solution vector) or if CVODES will evaluate the right-hand sides of the sensitivity systems, 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 $\bar{p}_{\text{plist}_i} = |p_i|$ can be used.

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

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

10. **Set sensitivity initial conditions**

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

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

    [**S**] `yS0 = N_VNewVectorArray_Serial(Ns, N);`

    [**P**] `yS0 = N_VNewVectorArray_Parallel(Ns, N);`

    and, for each $i = 1, \cdots$,`Ns`, load initial values for the i-th sensitivity vector into the structure defined by:

    [**S**] `NV_DATA_S(yS0[i])`

    [**P**] `NV_DATA_P(yS0[i])`

    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_VNewVectorArrayEmpty_Serial(Ns, N);`

    [**P**] `yS0 = N_VNewVectorArrayEmpty_Parallel(Ns, N);`

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]);`

[**P**] `N_VSetArrayPointer_Parallel(yS0_i, yS0[i]);`

11. **Activate sensitivity calculations**

    Call `flag = CVodeSensMalloc(...);` to activate forward sensitivity computations and allocate internal memory for CVODES related to sensitivity calculations (see §6.2.1).

12. **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.

13. **Advance solution in time**

14. **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 the routine `flag = CVodeGetSens(cvode_mem, t, yS);` (see §6.2.2).

15. **Deallocate memory for solution vector**

16. **Deallocate memory for sensitivity vectors**

    Upon completion of the integration, deallocate memory for the vectors `yS0`:

    [**S**] `N_VDestroyVectorArray_Serial(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`.

17. **Free user data structure**

18. **Free solver memory**

19. **Free vector specification memory**

6.2 User-callable routines for forward sensitivity analysis

This section describes the CVODES functions, additional to those presented in §5.5, that are called by the user to setup and solve a forward sensitivity problem.

6.2.1 Forward sensitivity initialization and deallocation functions

Activation of forward sensitivity computation is done by calling `CVodeSensMalloc`. The form of the call to this routine is as follows:

`CVodeSensMalloc`

Call `flag = CVodeSensMalloc(cvode_mem, Ns, ism, yS0);`

Description The routine `CVodeSensMalloc` 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 and 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 §6.3).

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 CVodeSensMalloc 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 CVodeSensMalloc has an illegal value.

Notes If an error occured, CVodeSensMalloc also prints an error message to the file specified by the optional input errfp.

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 by:

- Base value: lenrw = lenrw + (maxord+5)$N_s N$
- With itolS = CV_SV (see CVodeSetSensTolerances): lenrw = lenrw +$N_s N$

the size of the integer workspace is increased by:

- Base value: leniw = leniw + (maxord+5)$N_s N$
- With itolS = CV_SV: leniw = leniw +$N_s N$

The routine CVodeSensReInit, useful during the solution of a sequence of problems of same size, reinitializes the sensitivity-related internal memory and must follow a call to CVodeSensMalloc (and maybe a call to CVodeReInit). The number Ns of sensitivities is assumed to be unchanged since the call to CVodeSensMalloc. 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 and 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_SENSI` Memory space for sensitivity integration was not allocated through a previous call to `CVodeSensMalloc`.
- `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 `CVodeSensMalloc`.

If an error occured, `CVodeSensReInit` also prints an error message to the file specified by the optional input `errfp`.

To deallocate all forward sensitivity-related memory (allocated through a preious call to `CVodeSensMalloc`), the user must call

`CVodeSensFree`

Call `CVodeSensFree(cvode_mem);`

Description The function `CVodeSensFree` frees the memory allocated for forward sensitivity computaions by a previous call to `CVodeSensMalloc`.

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

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

Notes After a call to `CVodeSensFree`, forward sensitivity computations can be reactivated only by calling again `CVodeSensMalloc`.

To activate and deactivate forward sensitivity calculations for successive CVODES runs, without having to allocate and deallocate memory, the following function is provided:

`CVodeSensToggle`

Call `CVodeSensToggle(cvode_mem, sensi);`

Description The function `CVodeSensToggle` activates or deactivates forward sensitivity calculations. It does *not* deallocate sensitivity-related memory.

Arguments
- `cvode_mem` (`void *`) pointer to the memory previously allocated by `CVodeMalloc`.
- `sensi` (`booleantype`) flag indicating activation (`sensi = TRUE`) or deactivation (`sensi = FALSE`) of sensitivity computations.

Return value The return value `flag` of `CVodeSensToggle` is one of:

- `CV_SUCCESS` `CVodeGetSens` was successful.
- `CV_MEM_NULL` `cvode_mem` was `NULL`.
- `CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes It is allowed to set `sensi = TRUE` only if `CVodeSensMalloc` has been previously called to alocate sensitivity-related memory.

6.2.2 Forward sensitivity extraction functions

If forward sensitivity computations have been initialized by a call to `CVodeSensMalloc`, 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 `y`. Solution sensitivities can be obtained through one of the following functions:

`CVodeGetSens`

Call `flag = CVodeGetSens(cvode_mem, t, 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 `CVodeMalloc`.
- `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.
- `yS` (`N_Vector *`) the 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` `yQ` is `NULL`.
- `CV_BAD_T` The time `t` is not in the allowed range.

Notes In case of an error return, an error message is also printed.

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 `CVodeMalloc`.
- `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 *`) the vectors containing the derivatives. 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` The pointer to `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 $0, 1, ..., q_u$.
- `CV_BAD_T` The time `t` is not in the allowed range.

Notes In case of an error return, an error message is also printed.

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, t, 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 `CVodeMalloc`.
- `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.
- `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` `yQ` is `NULL`.
- `CV_BAD_T` The time `t` is not in the allowed range.

Notes In case of an error return, an error message is also printed.

`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 `CVodeMalloc`.
- `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 that is 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` One of the vectors `dkyS` is `NULL`.
- `CV_BAD_IS` The index `is` is not in the allowed range.
- `CV_BAD_K` `k` is not in the range $0, 1, ..., q_u$.
- `CV_BAD_T` The time `t` is not in the allowed range.

Notes In case of an error return, an error message is also printed.

6.2.3 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 6.1 lists all forward sensitivity optional input functions in CVODES which are described in detail in the remainder of this section.

Table 6.1: Forward sensitivity optional inputs

Optional input	Routine name	Default
Sensitivity right-hand side routine	CVodeSetSensRhsFn	internal DQ
Sensitivity right-hand side routine	CVodeSetSensRhs1Fn	internal DQ
Data for sensitivity right-hand side routine	CVodeSetSensFdata	NULL
Sensitivity scaling factors	CVodeSetSensPbar	NULL
Type of DQ approximation	CVodeSetSensRho	0.0
Error control strategy	CVodeSetSensErrCon	FALSE
Sensitivity integration tolerances	CVodeSetSensTolerances	estimated
Maximum no. of nonlinear iterations	CVodeSetSensMaxNonlinIters	3

`CVodeSetSensRhsFn`

Call `flag = CVodeSetSensRhsFn(cvode_mem, fS);`

Description The function `CVodeSetSensRhsFn` specifies the user-supplied C function used to evaluate the sensitivity right-hand sides (for all parameters at once).

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`fS` (`CVSensRhsFn`) user-defined sensitivity right-hand side function.

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

Notes This type of sensitivity right-hand side function is not compatible with the `CV_STAGGERED1` sensitivity solution method (argument `ism` to `CVodeSensMalloc`). The compatibility test is performed at the first step in `CVode`.

Passing `fS=NULL` indicates using the default internal difference quotient sensitivity right-hand side routine.

`CVodeSetSensRhs1Fn`

Call `flag = CVodeSetSensRhs1Fn(cvode_mem, fS);`

Description The function `CVodeSetSensRhs1Fn` specifies the user-supplied C function used to evaluate the sensitivity right-hand sides (one parameter at a time).

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`fS` (`CVSensRhs1Fn`) user-defined sensitivity right-hand side function.

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

Notes This type of sensitivity right-hand side function *must* be used when the `CV_STAGGERED1` sensitivity solution method is selected through `CVodeSensMalloc`, but can also be used with the other two choices (`CV_SIMULTANEOUS` and `CV_STAGGERED`).

Passing `fS=NULL` indicates using the default internal difference quotient sensitivity right-hand side routine.

`CVodeSetSensFdata`

Call `flag = CVodeSetSensFdata(cvode_mem, fS_data);`

Description The function `CVodeSetSensFdata` specifies the user data block for use by the user-supplied sensitivity right-hand side function and attaches it to the main CVODES memory block.

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`fS_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 `fS_data` is not specified, a `NULL` pointer is passed to the user-supplied sensitivity right-hand side function.

The pointer `fS_data` can be the same as the pointer `f_data` specified through `CVodeSetFdata` (see §5.5.4) and passed to the user-supplied right-hand side function f.

`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 *f_data* passed to the righ-hand side function. (See §6.1).
`pbar` (`realtype *`) an array of `Ns` positive scaling factors. If non-`NULL`, `pbar` must have all its components > 0.0. (See §6.1).
`plist` (`int *`) an array of `Ns` positive flags to specify which parameters to use in estimating the sensitivity equations. If non-`NULL`, `plist` must have all components ≥ 1. (See §6.1).

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

`CV_SUCCESS` The optional value has been successfully set.
`CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
`CV_ILL_INPUT` An argument has an illegal value.

Notes ! While `plist` and `pbar` have default values (see §6.1), an error will occur if neither `p` nor a user-provided sensitivity right-hand side function were not specified.

`CVodeSetSensRho`

Call `flag = CVodeSetSensRho(cvode_mem, rho);`

Description The function `CVodeSetSensRho` 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.
`rho` (`realtype`) value of the selection parameter.

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 `rho`$= 0.0$. See §3.2 for details.

`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 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, `errconQ` is set to `FALSE`. If `errcon=TRUE` then both state variables and sensitivity variables are included in the error tests. If `errcon=FALSE` then the sensitivity variables are excluded from the error tests. Note that, in any event, all variables are considered in the convergence tests.

`CVodeSetSensTolerances`

Call `flag = CVodeSetSensTolerances(cvode_mem, itolS, reltolS, abstolS);`

Description The function `CVodeSetSensTolerances` specifies the integration tolerances for sensitivity variables.

Arguments
- `cvode_mem` (`void *`) pointer to the CVODES memory block.
- `itolS` (`int`) is one of `CV_SS`, `CV_SV`, or `CV_EE`, where `itolS=CV_SS` indicates scalar relative error tolerance and scalar absolute error tolerance, while `itolS=CV_SV` indicates scalar relative error tolerance and vector absolute error tolerance. `itolS=CV_EE`, the arguments `reltolS` and `abstolS` are ignored and CVODES will estimate tolerances for the sensitivity variables based on the state tolerances and the scaling factors $\bar{p}$.
- `reltolS` (`realtype`) is the relative error tolerance.
- `abstolS` (`void *`) is a pointer to the absolute error tolerance. If `itolS=CV_SS`, `abstolS` must be a pointer to an array of `realtype` variables. If `itolS=CV_SV`, `abstolS` must be an array of `Ns` variables of type `N_Vector`. In the latter case, `abstolS` should be created and set in the same manner as the vectors of initial values for the sensitivity variables (see §6.1).

Return value The return value `flag` (of type `int`) is one of:
- `CV_SUCCESS` The optional values have been successfully set.
- `CV_MEM_NULL` The `cvode_mem` pointer is `NULL`.
- `CV_ILL_INPUT` An input argument to `CVodeSetSensTolerances` has an illegal value.

Notes The default behavior is for CVODES to estimated appropriate integration tolerances for the sensitivity variables based on the state tolerances and the scaling factors $\bar{p}$. See §3.2 for details.

! It is the user's responsibility to provide compatible `itolS` and `abstolS` arguments.

`CVodeSetSensMaxNonlinIters`

Call `flag = CVodeSetSensMaxNonlinIters(cvode_mem, maxcorS);`

Description The function `CVodeSetSensMaxNonlinIters` specifies the maximum number of nonlinear solver iterations for sensitivity variables per step.

Table 6.2: Forward sensitivity optional outputs

Optional output	Routine name
No. of calls to sensitivity r.h.s. function	CVodeGetNumSensRhsEvals
No. of calls to r.h.s. function for sensitivity	CVodeGetNumRhsEvalsSens
No. of sensitivity local error test failures	CVodeGetNumSensErrTestFails
No. of calls to lin. solv. setup routine for sens.	CVodeGetNumSensLinSolvSetups
Error weight vector for sensitivity variables	CVodeGetSensErrWeights
No. of sens. nonlinear solver iterations	CVodeGetNumSensNonlinSolvIters
No. of sens. convergence failures	CVodeGetNumSensNonlinSolvConvFails
No. of staggered nonlinear solver iterations	CVodeGetNumStgrSensNonlinSolvIters
No. of staggered convergence failures	CVodeGetNumStgrSensNonlinSolvConvFails

Arguments `cvode_mem` (`void *`) pointer to the CVODES memory block.
`maxcorS` (`int`) maximum number of nonlinear solver iterations allowed per step.

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.

6.2.4 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 6.2 and described in detail in the remainder of this section.

`CVodeGetNumSensRhsEvals`

Call `flag = CVodeGetNumSensRhsEvals(cvode_mem, &nfSevals);`

Description The function `CVodeGetNumSensRhsEvals` returns the number of calls to the sensitivity right-hand side function.

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

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

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

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

`CV_NO_SENS` Forward sensitivity analysis was not initialized.

Notes In order to accommodate any of the three possible sensitivity solution methods, the default internal finite difference quotient functions evaluate the sensitivity right-hand sides one at a time. Therefore, `nfSevals` will always be a multiple of the number of sensitivity parameters (the same as the case in which the user supplies a routine of type `CVSensRhs1Fn`).

`CVodeGetNumRhsEvalsSens`

Call `flag = CVodeGetNumRhsEvalsSens(cvode_mem, &nfevalsS);`

Description The function `CVodeGetNumRhsEvalsSEns` returns the number of calls to the user's right-hand side function due to the internal finite difference approximation of the sensitivity right-hand sides.

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

`nfevalsS` (`long int`) number of calls to the user 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 This counter is incremented only if the internal finite difference approximation routines are used for the evaluation of the sensitivity right-hand sides.

`CVodeGetNumSensErrTestFails`

Call `flag = CVodeGetNumSensErrTestFails(cvode_mem, &nSetfails);`

Description The function `CVodeGetNumSensErrTestFails` returns the number of local error test failures for the sensitivity variables that have occured.

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 §6.2.3). Even in that case, this counter is not incremented if the `ism=CV_SIMULTANEOUS` sensitivity solution method has been used.

`CVodeGetNumSensLinSolvSetups`

Call `flag = CVodeGetNumSensLinSolvSetups(cvode_mem, &nlinsetupsS);`

Description The function `CVodeGetNumSensLinSolvSetups` 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 in the call to `CVodeSensMalloc` (see §6.2.1).

`CVodeGetSensStats`

Call
```
flag = CVodeGetSensStats(cvode_mem, &nfSevals, &nfevalsS,
                                    &nSetfails, &nlinsetupsS);
```

Description The function `CVodeGet` 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 user-supplied right-hand side function.

`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 weights at the current time. These are the reciprocals of the W_i of (3.7) for the sensitivity variables.

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

eSweight (N_Vector_S) 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 need not allocate space for eSweight and should not modify any of its components.

`CVodeGetNumSensNonlinSolvIters`

Call `flag = CVodeGetNumSensNonlinSolvIters(cvode_mem, &nSniters);`

Description The function CVodeGetNumSensNonlinSolvIters returns the number of nonlinear (functional or Newton) 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 the ism was CV_STAGGERED or CV_STAGGERED1 in the call to CVodeSensMalloc (see §6.2.1).

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 CVodeGetNumStgrSensNonlinSolvIters (see below).

`CVodeGetNumSensNonlinSolvConvFails`

Call `flag = CVodeGetNumSensNonlinSolvConvFails(cvode_mem, &nSncfails);`

Description The function CVodeGetNumSensNonlinSolvConvFails 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 the ism was CV_STAGGERED or CV_STAGGERED1 in the call to CVodeSensMalloc (see §6.2.1).

In the CV_STAGGERED1 case, the value of nSncfails is the sum of the number of nonlinear convergence failures that occured for each sensitivity equation. These individual counters can be obtained through a call to CVodeGetNumStgrSensNonlinConvFails (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.

CVodeGetNumStgrSensNonlinSolvIters

Call: `flag = CVodeGetNumStgrSensNonlinSolvIters(cvode_mem, nSTGR1niters);`

Description: The function CVodeGetNumStgrSensNonlinSolvIters 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.

CVodeGetNumStgrSensNonlinSolvConvFails

Call: `flag = CVodeGetNumStgrSensNonlinSolvConvFails(cvode_mem, nSTGR1ncfails);`

Description: The function CVodeGetNumStgrSensNonlinSolvConvFails 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.

6.3 User-supplied routines for forward sensitivity analysis

In addition to the required and optional user-supplied routines described in §5.6, 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 (3.9).

By default, CVODES uses difference quotient approximation routines for the right-hand sides of the sensitivity equations. However, CVODES allows the option for user-defined sensitivity right-hand side routines (which also provides a mechanism for interfacing CVODES to routines generated by automatic differentiation).

Sensitivity equations right-hand side (all at once)

If the CV_SIMULTANEOUS or CV_STAGGERED approach was selected in the call to CVodeSensMalloc, the user may provide the right-hand sides of the sensitivity equations (3.9), for all sensitivity parameters at once, through a function of type CVSensRhsFn defined by:

CVSensRhsFn

Definition
```
typedef void (*CVSensRhsFn)(int Ns, realtype t,
                            N_Vector y, N_Vector ydot,
                            N_Vector *yS, N_Vector *ySdot,
                            void *fS_data,
                            N_Vector tmp1, N_Vector tmp2);
```

Purpose This function computes the sensitivity right-hand side for all sensitivity equations at once. It must compute the vectors $(\partial f/\partial y)s_i(t) + (\partial f/\partial p_i)$ and store them in ySdot[i].

Arguments

t	is the current value of the independent variable.
y	is the current value of the state vector, $y(t)$.
ydot	is the current value of the right-hand side of the state equations.
yS	contains the current values of the sensitivity vectors.
ySdot	is the output of CVSensRhsFn. On exit it must contain the sensitivity right-hand side vectors.
f_data	is a pointer to user data - the same as the fS_data parameter passed to CVodeSetSensFdata.
tmp1	
tmp2	are N_Vectors which can be used as temporary storage.

Return value A CVSensRhsFn function type does not have a return value.

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.

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 `CVodeSensMalloc` argument `ism`, and is required if `ism`=`CV_STAGGERED1`. The type `CVSensRhs1Fn` is defined by

`CVSensRhs1Fn`

Definition

```
typedef void (*CVSensRhs1Fn)(int Ns, realtype t,
                             N_Vector y, N_Vector ydot,
                             int iS, N_Vector yS, N_Vector ySdot,
                             void *fS_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.
- `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.
- `f_data` is a pointer to user data - the same as the `fS_data` parameter passed to `CVodeSetSensFdata`.
- `tmp1`
- `tmp2` are `N_Vector`s which can be used as temporary storage.

Return value A `CVSensRhs1Fn` function type does not have a return value.

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

6.4 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 `cvdx` and the second run of the `cvfdx` example in [17] 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 §6.2.1, even with partial error control selected in the call to `CVodeSensMalloc`, the sensitivity variables are included in the convergence tests of the nonlinear solver.

When using the simultaneous corrector method (§3.2), 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 `ism=CV_STAGGERED1` in the call to `CVodeSensMalloc`), 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 CVSPGMR). 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 correction), 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 7

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 (3.17) or (3.20), 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 Chapter 11.

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 §5.

7.1 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 §5.4, 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 with [**P**] correspond to NVECTOR_PARALLEL, while steps marked with [**S**] correspond to NVECTOR_SERIAL.

1. **Include necessary header files**

   Besides `cvodes.h`, the main program must also include `cvodea.h` which defines additional types and constants, and includes function prototypes for the adjoint sensitivity module user-callable functions and `cvodes.h`. In addition, the main program should include an NVECTOR implementation header file (`nvector_serial.h` or `nvector_parallel.h` for the two implementations provided with CVODES) and, if Newton iteration was selected, a header file from 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. **Set optional inputs for the forward problem**

8. **Attach linear solver module for the forward problem**

9. **Set linear solver optional inputs for the forward problem**

10. **Allocate space for the adjoint computation**

    Call `cvadj_mem = CVadjMalloc()` to allocate memory for the combined forward-backward problem (see §7.2.1 for more details). This call requires `Nd`, the number of steps between two consecutive checkpoints.

11. **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 §7.2.2)). The final value of `tret`, denoted `tfinal`, is then the maximum allowable value for the endpoint t_1.

Backward problem

12. **Set problem dimensions for the backward problem**

    [**S**] set `NB`, the number of variables in the backward problem
    [**P**] set `NB` and `NBlocal`

13. **Set final values for the backward problem**

    Set the vector `yB0` of final values for the backward problem.

14. **Create CVODES object for the backward problem**

    Call `CVodeCreateB`, a wrapper for `CVodeCreate`, to create the CVODES memory block and specify the solution method (linear multistep method and nonlinear solver iteration type) for the backward problem. Unlike `CVodeCreate`, the function `CVodeCreateB` does not return a pointer to the newly created memory block. Instead, this pointer is attached to the adjoint memory block (returned by `CVadjMalloc` and passed as the first argument to `CVodeCreateB`).

15. **Allocate memory for the backward problem**

    Call `CVodeMallocB`, a wrapper for `CVodeMalloc`, to allocate internal memory and initialize CVODES at `tB0` for the backward problem (see §7.2.3).

16. **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 as their first argument the adjoint memory block returned by `CVadjMalloc`.

17. **Attach linear solver module for the backward problem**

    If Newton iteration is chosen, initialize the linear solver module for the backward problem by calling the appropriate wrapper function: `CVDenseB`, `CVBandB`, `CVDiagB`, or `CVSpgmrB` (see §7.2.4). 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.

18. **Initialize quadrature calculation**

    If additional quadrature equations must be evaluated , call `CVodeQuadMallocB`, a wrapper for `CVodeQuadMalloc`, 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.

19. **Integrate backward problem**

    Call `CVodeB`, a second wrapper for the CVODES main integration function `CVode`, to integrate the backward problem from `tB0` (see §7.2.5). 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 of the forward problem.

20. **Extract quadrature variables**

    If applicable, call `CVodeGetQuadB`, a wrapper for `CVodeGetQuad`, to extract the values of the quadrature variables at the time returned by the last call to `CVodeB`.

21. **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, and a call to `CVadjFree` (see §7.2.1) to free the memory allocated for the combined problem. Note that `CVadjFree` also deallocates the CVODES memory for the backward problem.

22. **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 (12)-(20) are not present, a program with the above structure will have the same functionality as one described in §5.4 for integration of ODEs, albeit with some overhead due to the checkpointing scheme.

7.2 User-callable functions for adjoint sensitivity analysis

7.2.1 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 `CVadjMalloc`. The form of the call to this function is

`CVadjMalloc`

Call `cvadj_mem = CVadjMalloc(cvode_mem, Nd);`

Description The function `CVadjMalloc` allocates internal memory for the combined forward and backward integration, other than the CVODES memory block. Space is allocated for the N_d interpolation data points and a linked list of checkpoints is initialized.

Arguments
- `cvode_mem` (`void *`) is the CVODES memory block for the forward problem returned by a previous call to `CVodeCreate`.
- `Nd` (`long int`) is the number of integration steps between two consecutive checkpoints.

Return value If successful, `CVadjMalloc` returns a pointer of type `void *`. The user does not need to access this memory block but must pass it to other adjoint module user-callable functions. In case of failure (`cvode_mem` is `NULL`, `Nd` is nonpositive, or a memory request fails), `CVadjMalloc` prints an error message to the standard output stream `stderr` and returns `NULL`.

Notes The user must set `Nd` so that all data needed for interpolation of the forward problem solution between two checkpoints fits in memory. `CVadjMalloc` attempts to allocate space for (2Nd+3) variables of type `N_Vector`.

`CVadjFree`

Call `CVadjFree(cvadj_mem);`

Description The function `CVadjFree` frees the memory allocated by a previous call to `CVadjMalloc`.

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

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

Notes This function frees all memory allocated by `CVadjMalloc`. This includes workspace memory, the linked list of checkpoints, memory for the interpolation data, the CVODES memory for the backward integration phase, and possibly memory for backward-phase preconditioners.

7.2.2 Forward integration function

The function `CVodeF` is very similar to the CVODES function `CVode` (see §5.5.3) in that it integrates the solution of the forward problem and returns the solution in `y`. At the same time, however, `CVodeF` stores checkpoint data every `Nd` integration steps. `CVodeF` can be called repeatedly by the user. The call to this function has the form

`CVodeF`

Call `flag = CVodeF(cvadj_mem, tout, yout, tret, itask, ncheck);`

Description The function `CVodeF` integrates the forward problem over an interval in t and saves checkpointing data.

Arguments
- `cvadj_mem` (`void *`) pointer to the adjoint 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.
- `itask` (`int`) a flag indicating the job of the solver for the next step. The `CV_NORMAL` task is to have the solver take internal steps until it has reached or just passed the user-specified `tout` parameter. The solver then interpolates in order to return an approximate value of y(`tout`). The `CV_ONE_STEP` option tells the solver to just take one internal step and return the solution at the point reached by that step. The `CV_NORMAL_TSTOP` and `CV_ONE_STEP_TSTOP` modes are similar to `CV_NORMAL` and `CV_ONE_STEP`, respectively, except that the integration never proceeds past the value `tstop` (specified through the function `CVodeSetStopTime`).

Return value On return, `CVodeF` returns a vector `yout` and a corresponding independent variable value t =`*tret`, such that `yout` is the computed value of $y(t)$. Additionally, it returns in `ncheck` the number of checkpoints saved.

The return value `flag` (of type `int`) will be one of the following. For more details see §5.5.3.

- `CV_SUCCESS` `CVodeF` succeeded.
- `CV_TSTOP_RETURN` `CVodeF` succeeded by reaching the optional stopping point.
- `CV_NO_MALLOC` The function `CVodeMalloc` has not been previously called.

CV_ILL_IPUT	One of the inputs to CVodeF is illegal.
CV_TOO_MUCH_WORK	The solver took mxstep internal steps but could not reach tout.
CV_TOO_MUCH_ACC	The solver could not satisfy the accuracy demanded by the user for some internal step.
CV_ERR_FAILURE	Error test failures occurred too many times during one internal time step or occurred with $\|h\| = h_{min}$.
CV_CONV_FAILURE	Convergence test failures occurred too many times during one internal time step or occurred with $\|h\| = h_{min}$.
CV_LSETUP_FAIL	The linear solver's setup function failed in an unrecoverable manner.
CV_LSOLVE_FAIL	The linear solver's solve function failed in an unrecoverable manner.
CV_ADJMEM_NULL	The cvadj_mem argument was NULL.
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, information for the Hermite interpolation is already available from the last checkpoint forward. In particular, if no check points were necessary, there is no need for the second forward integration phase.

! Although it is legal to define a value for tstop and then call CVodeF with itask = CV_NORMAL_TSTOP or itask = CV_ONE_STEP_TSTOP, after a return with flag = TSTOP_RETURN, the integration should not be continued (no tstop information is stored at checkpoints).

! It is illegal to change the integration tolerances between consecutive calls to CVodeF, as this information is not captured in the checkpoints data.

7.2.3 Backward problem initialization functions

The functions CVodeCreateB and CVodeMallocB 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(cvadj_mem, lmm, iter);`

Description: The function CVodeCreateB instantiates a CVODES solver object and specifies the solution method for the backward problem.

Arguments:

cvadj_mem	(void *) pointer to the adjoint memory block returned by CVadjMalloc.
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.

Return value If successful, CVodeCreateB stores a pointer to the newly created CVODES memory block (of type void *) for the backward problem. The return value flag (of type int) is one of:

CV_SUCCESS The call to CVodeCreateB was successful.

CV_ADJMEM_NULL The cvadj_mem argument was NULL.

CV_MEM_FAIL An error occured while trying to create the CVODES memory block for the backward problem.

The function CVodeMallocB is essentially a call to CVodeMalloc with some particularization for backward integration as described below.

CVodeMallocB

Call `flag = CVodeMallocB(cvadj_mem, fB, tB0, yB0, itolB, reltolB, abstolB);`

Description The function CVodeMallocB provides required problem and solution specifications, allocates internal memory, and initializes CVODES for the backward problem.

Arguments cvadj_mem (void *) pointer to the adjoint memory block returned by CVadjMalloc.

fB (CVRhsFnB) is the C function which computes fB, the right-hand side of the backward ODE problem. This function has the form fB(t, y, yB, yBdot, f_dataB) (for full details see §7.3).

tB0 (realtype) specifies the endpoint where final conditions are provided for the backward problem.

yB0 (N_Vector) is the final value of the backward problem.

itolB (int) is one of CV_SS or CV_SV where itol=CV_SS indicates scalar relative error tolerance and scalar absolute error tolerance, while itol=CV_SV indicates scalar relative error tolerance and vector absolute error tolerance.

reltolB (realtype) is the relative error tolerance.

abstolB (void *) is a pointer to the absolute error tolerance. If itolB=CV_SS, abstolB must be a pointer to a realtype variable. If itolB=CV_SV, abstolB must be an N_Vector variable.

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

CV_SUCCESS The call to CVodeMallocB was successful.

CV_ADJMEM_NULL The cvadj_mem argument was NULL.

CV_BAD_TB0 The final time tB0 is outside the interval over which the forward problem was solved.

CV_MEM_NULL The CVODES memory block for the backward problem was not initialized through a previous call to CVodeCreateB.

CV_MEM_FAIL A memory allocation request has failed.

CV_ILL_INPUT An input argument to CVodeMallocB has an illegal value.

Notes ! It is the user's responsibility to provide compatible itolB and abstolB arguments.

For the case when it is needed to solve several different backward problems corresponding to the same original problem, CVODES provides a mechanism to reuse the existing checkpoints. The function CVodeReInitB reinitializes the CVODES memory block for the backward problem, where a prior call to CVodeMallocB has been made with the same problem size NB. CVodeReInitB performs the same input checking and initializations that CVodeMallocB does, but does no memory allocation, assuming that the existing internal memory is sufficient for the new problem. Note that CVReInitB is essentially a wrapper for CVodeReInit and so all details given for CVodeReInit in §5.5.7 apply. The call to the CVodeReInitB function has the form

`CVodeReInitB`

Call `flag = CVodeReInitB(cvadj_mem, fB, tB0, yB0, itolB, reltolB, abstolB);`

Description The function `CVodeReInitB` provides required problem specifications and reinitializes CVODES for the backward problem.

Arguments
- `cvadj_mem` (`void *`) pointer to the adjoint memory block returned by `CVadjMalloc`.
- `fB` (`CVRhsFnB`) is the C function which computes fB, the right-hand side of the backward ODE problem. This function has the form `fB(t, y, yB, yBdot, f_dataB)` (for full details see §7.3).
- `tB0` (`realtype`) specifies the endpoint where final conditions are provided for the backward problem.
- `yB0` (`N_Vector`) is the final value of the backward problem.
- `itolB` (`int`) is either `CV_SS` or `CV_SV`, where `itol=CV_SS` indicates scalar relative error tolerance and scalar absolute error tolerance, while `itol=CV_SV` indicates scalar relative error tolerance and vector absolute error tolerance.
- `reltolB` (`realtype`) is the relative error tolerance.
- `abstolB` (`void *`) is a pointer to the absolute error tolerance. If `itolB=CV_SS`, `abstolB` must be a pointer to a `realtype` variable. If `itolB=CV_SV`, `abstolB` must be an `N_Vector` variable.

Return value The return value `flag` (of type `int`) will be one of the following:
- `CV_SUCCESS` The call to `CVodeReInitB` was successful.
- `CV_ADJMEM_NULL` The `cvadj_mem` argument was `NULL`.
- `CV_BAD_TB0` The final time `tB0` is outside the interval over which the forward problem was solved.
- `CV_MEM_NULL` The CVODES memory block for the backward problem was not initialized through a previous call to `CVodeCreateB`.
- `CV_NO_MALLOC` Memory space for the CVODES memory block for the backward problem was not allocated through a previous call to `CVodeMallocB`.
- `CV_ILL_INPUT` An input argument to `CVodeReInitB` has an illegal value.

Notes ! It is the user's responsibility to provide compatible `itolB` and `abstolB` arguments.

Note that memory allocated by `CVodeMallocB` is deallocated by the function `CVadjFree`.

7.2.4 Linear solver initialization functions for backward problem

The adjoint sensitivity module in CVODES provides interfaces to the initialization functions of all supported linear solver modules for the case in which Newton iteration is selected for the solution of the backward problem. The initialization functions described in §5.5.2 cannot be directly used since the optional user-defined Jacobian-related functions have different prototypes for the backward problem than for the forward problem.

The following four 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 §5.5.2 with the exception of their first argument which must be the pointer to the adjoint memory block returned by `CVadjMalloc`.

```
flag = CVDenseB(cvadj_mem, nB);
flag = CVDiagB(cvadj_mem);
flag = CVBandB(cvadj_mem, nB, mupperB, mlowerB);
flag = CVSpgmrB(cvadj_mem, pretypeB, maxlB);
```

Their return value `flag` (of type `int`) can have any of the return values of their counterparts, but it can also be `CV_ADJMEM_NULL` if the `cvadj_mem` argument was `NULL`.

7.2.5 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 integrations between consecutive checkpoints. The first run integrates the original IVP forward in time and stores interpolation data; the second run integrates the backward problem backward in time and performs Hermite interpolation to provide the solution of the IVP to the backward problem.

The call to this function has the form

CVodeB

Call `flag = CVodeB(cvadj_mem, tBout, yBout, tBret, itaskB);`

Description The function CVodeB integrates the backward ODE problem over an interval in t.

Arguments
- cvadj_mem (void *) pointer to the adjoint memory block returned by CVadjMalloc.
- tBout (realtype) the next time at which a computed solution is desired.
- yBout (N_Vector) the computed solution vector of the backward problem.
- tBret (realtype *) the time reached by the solver.
- 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 tBout parameter. The solver then interpolates in order to return an approximate value of yB(tBout). The CV_ONE_STEP option tells the solver to just take one internal step and return the solution at the point reached by that step.

Return value On return, CVodeB returns a vector yBout and a corresponding independent variable value $t =$*tBret, such that yBout is the computed value of the solution of the backward problem.

The return value flag (of type int) will be one of the following. For more details see §5.5.3.

CV_SUCCESS	CVodeB succeeded.
CV_NO_MALLOC	The CVODES memory for the backward problem was NULL.
CV_ILL_INPUT	One of the inputs to CVode 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_ADJMEM_NULL	The cvadj_mem argument was NULL.
CV_BCKMEM_NULL	The cvodes memory for the backward problem was not created through 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 occured 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.

7.2.6 Optional input and output functions for the backward problem

Optional input functions

The adjoint module in CVODES provides wrappers for most of the optional input functions defined in §5.5.4. The only difference is that the first argument of the optional input functions for the backward problem is the pointer to the adjoint memory block, cvadj_mem of type void *, returned by CVadjMalloc. The optional input functions defined for the backward problem are:

```
flag = CVodeSetIterTypeB(cvadj_mem, iterB);
flag = CVodeSetFdataB(cvadj_mem, f_dataB);
flag = CVodeSetErrFileB(cvadj_mem, errfpB);
flag = CVodeSetMaxOrdB(cvadj_mem, maxordB);
flag = CVodeSetMaxNumStepsB(cvadj_mem, mxstepsB);
flag = CVodeSetStabLimDetB(cvadj_mem, stldetB);
flag = CVodeSetInitStepB(cvadj_mem, hinB);
flag = CVodeSetMinStepB(cvadj_mem, hminB);
flag = CVodeSetMaxStepB(cvadj_mem, hmaxB);
```

Their return value flag (of type int) can have any of the return values of their counterparts, but it can also be CV_ADJMEM_NULL if the cvadj_mem argument was NULL.

Optional inputs for the CVDENSE linear solver module can be set for the backward problem through the following function:

CVDenseSetJacFnB

Call: flag = CVDenseSetJacFnB(cvadj_mem, djacB, jac_dataB);

Description: The function CVDenseSetJacFnB specifies the dense Jacobian approximation function to be used for the backward problem and the pointer to user data.

Arguments:

cvadj_mem	(void *) pointer to the adjoint memory block.
djacB	(CVDenseJacFnB) user-defined dense Jacobian approximation function.
jac_dataB	(void *) pointer to the user-defined data structure.

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

CVDENSE_SUCCESS	The optional value has been successfuly set.
CVDENSE_MEM_NULL	The CVODES solver memory block was not created through a call to CVodeCreateB.
CVDENSE_LMEM_NULL	The CVDENSE linear solver has not been initialized through a call to CVDenseB.
CV_ADJMEM_NULL	The cvadj_mem argument was NULL.

Notes: The function type CVDenseJacFnB is described in §7.3.

Optional inputs for the CVBAND linear solver module can be set for the backward problem through the following function:

CVBandSetJacFnB

Call: flag = CVBandSetJacFnB(cvadj_mem, bjacB, jac_dataB);

Description: The function CVBandSetJacFnB specifies the banded Jacobian approximation function to be used for the backward problem and the pointer to user data.

Arguments: cvadj_mem (void *) pointer to the adjoint memory block.

bjacB	(CVBandJacFnB) user-defined banded Jacobian approximation function.
jac_dataB	(void *) pointer to the user-defined data structure.

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

CVBAND_SUCCESS	The optional value has been successfuly set.
CVBAND_MEM_NULL	The CVODES solver memory block was not created through a call to CVodeCreateB.
CVBAND_LMEM_NULL	The CVBAND linear solver has not been initialized through a call to CVBandB.
CV_ADJMEM_NULL	The cvadj_mem argument was NULL.

Notes The function type CVBandJacFnB is described in §7.3.

Optional inputs for the CVSPGMR linear solver module can be set for the backward problem through the following functions:

CVSpgmrSetPreconditionerB

Call `flag = CVSpgmrSetPrecSolveFnB(cvadj_mem, psolveB, psetupB, p_dataB);`

Description The function CVSpgmrSetPrecSolveFnB specifies the preconditioner setup and solve functions and the pointer to user data for the backward integration.

Arguments

cvadj_mem	(void *) pointer to the adjoint memory block.
psolveB	(CVSpgmrPrecSolveFnB) user-defined preconditioner solve function.
psetupB	(CVSpgmrPrecSetupFnB) user-defined preconditioner setup function.
p_dataB	(void *) pointer to the user-defined data structure.

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

CVSPGMR_SUCCESS	The optional value has been successfuly set.
CVSPGMR_MEM_NULL	The CVODES solver memory block was not created through a call to CVodeCreateB.
CVSPGMR_LMEM_NULL	The CVSPGMR linear solver has not been initialized through a call to CVSpgmrB.
CV_ADJMEM_NULL	The cvadj_mem argument was NULL.

Notes The function type CVSpgmrPrecSolveFnB is described in §7.3. The function type CVSpgmrPrecSetupFnB is described in §7.3.

CVSpgmrSetJacTimesVecFnB

Call `flag = CVSpgmrSetJacTimesVecFnB(cvadj_mem, jtimesB, jac_data);`

Description The function CVSpgmrSetJacTimesFnB specifies the Jacobian-vector product function to be used and te pointer to user data.

Arguments

cvadj_mem	(void *) pointer to the adjoint memory block.
jtimesB	(CVSpgmrJacTimesVecFnB) user-defined Jacobian-vector product function.
jac_dataB	(void *) pointer to the user-defined data structure.

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

CVSPGMR_SUCCESS	The optional value has been successfuly set.
CVSPGMR_MEM_NULL	The CVODES solver memory block was not created through a call to CVodeCreateB.
CVSPGMR_LMEM_NULL	The CVSPGMR linear solver has not been initialized through a call to CVSpgmrB.
CV_ADJMEM_NULL	The cvadj_mem argument was NULL.

Notes The function type CVSpgmrJacTimesVecFnB is described in §7.3.

CVSpgmrSetGSTypeB

Call `flag = CVSpgmrSetGSType(cvadj_mem, gstypeB);`

Description The function `CVSpgmrSetGSTypeB` specifies the type of Gram-Schmidt orthogonalization to be used. 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 `cvadj_mem` (`void *`) pointer to the adjoint memory block.
`gstypeB` (`int`) type of Gram-Schmidt orthogonalization.

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

`CVSPGMR_SUCCESS`	The optional value has been successfuly set.
`CVSPGMR_MEM_NULL`	The CVODES solver memory block was not created through a call to `CVodeCreateB`.
`CVSPGMR_LMEM_NULL`	The CVSPGMR linear solver has not been initialized through a call to `CVSpgmrB`.
`CVSPGMR_ILL_INPUT`	The Gram-Schmidt orthogonalization type `gstypeB` is not valid.
`CV_ADJMEM_NULL`	The `cvadj_mem` argument was `NULL`.

Notes The default value is `MODIFIED_GS`.

CVSpgmrSetDeltB

Call `flag = CVSpgmrSetDeltB(cvadj_mem, deltB);`

Description The function `CVSpgmrSetDeltB` specifies the factor by which the GMRES convergence test constant is reduced from the Newton iteration test constant.

Arguments `cvadj_mem` (`void *`) pointer to the adjoint memory block.
`delt` (`realtype`) the value of the convergence test constant reduction factor.

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

`CVSPGMR_SUCCESS`	The optional value has been successfuly set.
`CVSPGMR_MEM_NULL`	The CVODES solver memory block was not created through a call to `CVodeCreateB`.
`CVSPGMR_LMEM_NULL`	The CVSPGMR linear solver has not been initialized through a call to `CVSpgmrB`.
`CVSPGMR_ILL_INPUT`	The factor `deltB` is negative.
`CV_ADJMEM_NULL`	The `cvadj_mem` argument was `NULL`.

Notes The default value is 0.05. Passing a value `deltB`= 0.0 also indicates using the default value.

CVSpgmrSetPrecTypeB

Call `flag = CVSpgmrSetPrecTypeB(cvadj_mem, pretypeB);`

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

Arguments `cvadj_mem` (`void *`) pointer to the adjoint memory block.
`pretypeB` (`int`) specifies the type of prconditioning and may be `PREC_NONE`, `PREC_LEFT`, `PREC_RIGHT`, or `PREC_BOTH`.

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

`CVSPGMR_SUCCESS`	The optional value has been successfuly set.
`CVSPGMR_MEM_NULL`	The CVODES solver memory block was not created through a call to `CVodeCreateB`.

CVSPGMR_LMEM_NULL	The CVSPGMR linear solver has not been initialized through a call to CVSpgmrB.
CVSPGMR_ILL_INPUT	The preconditioner type pretype is not valid.
CV_ADJMEM_NULL	The cvadj_mem argument was NULL.

Notes The preconditioning type is initially specified in the call to CVSpgmrB (see §7.2.4). This function call is needed only if pretypeB is being changed from its value in the previous call to CVSpgmrB.

Optional output functions

The user of the adjoint module in CVODES has access to any of the optional output functions described in §5.5.6, both for the main solver and for the linear solver modules. The first argument of these CVodeGet* functions is 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 a pointer to this memory block:

`CVadjGetCVodeBmem`

Call `cvode_memB = CVadjGetCVodeBmem(cvadj_mem);`

Description The function CvadjGetCVodeBmem returns a pointer to the CVODES memory block for the backward problem.

Arguments The argument cvadj_mem (of type void *) is a pointer to the adjoint memory block returned by CVadjMalloc.

Return value The return value, cvode_memB (of type void *), is a pointer to the CVODES memory for the backward problem.

Notes The user should not modify in any way cvode_memB.

7.2.7 Backward integration of pure quadrature equations

Backward quadrature initialization functions

The function CVodeQuadMallocB initializes and allocates memory for the backward integration of quadrature equations. It has the following form:

`CVodeQuadMallocB`

Call `flag = CVodeQuadMallocB(cvadj_mem, fQB, yQB0);`

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

Arguments

cvadj_mem	(void *) pointer to the adjoint memory block returned by CVadjMalloc.
fQB	(CVQuadRhsFnB) is the C function which computes fQB, the right-hand side of the backward quadrature equations. This function has the form fQB(t, y, yB, qBdot, fQ_dataB) (for full details see below).
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 CVodeQuadMallocB was successful.
CV_MEM_NULL	The CVODES solver memory block was not created through a previous call to CVodeCreateB.
CV_MEM_FAIL	A memory allocation request has failed.
CV_ADJMEM_NULL	The cvadj_mem argument was NULL.

The integration of quadrature equations during the backward phase can be re-initialized by calling

`CVodeQuadReInitB`

Call `flag = CVodeQuadReInitB(cvadj_mem, fQB, yQB0);`

Description The function `CVodeQuadReInitB` provides required problem specifications and reinitializes the backward quadrature integration.

Arguments
- `cvadj_mem` (`void *`) pointer to the adjoint memory block.
- `fQB` (`CVQuadRhsFnB`) is the C function which computes fQB, the right-hand side of the backward quadrature equations.
- `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 `CVodeReInitB` was successful.
- `CV_MEM_NULL` The CVODES solver memory block was not created through a previous call to `CVodeCreateB`.
- `CV_NO_QUAD` Quadrature integration was not activated through a previous call to `CVodeQuadMallocB`.
- `CV_ADJMEM_NULL` The `cvadj_mem` argument was `NULL`.

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 §5.7.2). The call to this function has the form

`CVodeGetQuadB`

Call `flag = CVodeGetQuadB(cvadj_mem, yQB);`

Description The function `CVodeGetQuadB` returns the quadrature solution vector after a successful return from `CVodeB`.

Arguments
- `cvadj_mem` (`void *`) pointer to the adjoint memory returned by `CVadjMalloc`.
- `yQB` (`N_Vector`) the computed backward quadrature vector.

Return value The return value `flag` of `CVodeGetQuadB` is one of:
- `CV_SUCCESS` `CVodeGetQuadB` was successful.
- `CV_MEM_NULL` The CVODES solver memory block was not created through a previous call to `CVodeCreateB`.
- `CV_NO_QUAD` Quadrature integration was not initialized through a previous call to `CVodeQuadMallocB`..
- `Cv_BAD_DKY` `yQB` is `NULL`.
- `CV_ADJMEM_NULL` The `cvadj_mem` argument was `NULL`.

Optional input and 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 §5.7.3:

```
flag = CVodeSetQuadFdataB(cvadj_mem, fQ_dataB);
flag = CVodeSetQuadErrConB(cvadj_mem, errconQB, itolQB, reltolQB, abstolQB);
```

Their return value `flag` (of type `int`) can have any of the return values of their counterparts, but it can also be `CV_ADJMEM_NULL` if the `cvadj_mem` argument was `NULL`.

Access to optional outputs related to backward quadrature integration can be obtained by calling the corresponding `CVodeGetQuad*` functions (see §5.7.4). A pointer 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 `CVadjGetCVodeBmem` (see §7.2.6).

7.2.8 Check Point Listing Function

For debugging purposes, CVODES provides a function `CVadjGetCheckPointsList` which displays partial information from the linked list of checkpoints generated by `CVodeF`. The call to this function has the form:

`CVadjGetCheckPointsList`

Call `CVadjCheckPointsList(cvadj_mem);`

Description The function `CVadjGetCheckPointsList` prints the memory addresses of the checkpoints and the time points at which the checkpoints were defined.

Arguments The argument `cvadj_mem` (of type `void *`) is a pointer to the adjoint memory block returned by `CVadjMalloc`.

Return value This function has no return value.

Notes For a typical output of `CVadjGetCheckPointsList`, see [17].

7.3 User-supplied functions for adjoint sensitivity analysis

In addition to the required ODE right-hand side function and any optional functions for the forward problem, when using the adjoint sensitivity module in CVODES, the user must supply one function defining the backward problem ODE and, optionally, functions to supply Jacobian-related information (if Newton iteration is chosen) and one or two functions that define the preconditioner (if the CVSPGMR solver is selected) for the backward problem. Type definitions for all these user-supplied functions are given below.

ODE right-hand side for the backward problem

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

`CVRhsFnB`

Definition
```
typedef void (*CVRhsFnB)(realtype t, N_Vector y, N_Vector yB,
                         N_Vector yBdot, void *f_dataB);
```

Purpose This function computes the right-hand side of the backward problem ODE system. This could be (3.17) or (3.20).

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 dependent variable vector.
- `yBdot` is the output vector containing the right-hand side of the backward ODE problem.
- `f_dataB` is a pointer to user data - the same as the `f_dataB` parameter passed to `CVodeSetFdataB`.

Return value A `CVRhsFnB` function type does not have a return value.

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

The `y`, `yB`, and `yBdot` arguments are all of type `N_Vector`, but `yB` and `yBdot` typically have different internal representations from `y`. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each NVECTOR implementation). For the sake of computational efficiency, the vector functions in the two NVECTOR implementations provided with CVODES do not perform any consistency checks with respect to their `N_Vector` arguments (see §8.1 and §8.2).

The `f_dataB` pointer is passed to the user's `fB` function every time it is called and can be the same as the `f_data` pointer used for the forward problem.

Quadrature right-hand side for the backward problem

The user must provide a function of type `CVQuadRhsFnB` defined by

`CVQuadRhsFnB`

Definition
```
typedef void (*CVQuadRhsFnB)(realtype t, N_Vector y, N_Vector yB,
                             N_Vector qBdot, void *fQ_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 dependent variable vector.
- `qBdot` is the output vector containing the right-hand side of the backward quadrature equations.
- `fQ_dataB` is a pointer to user data - the same as the `fQ_dataB` parameter passed to `CVodeSetQuadFdataB`.

Return value A `CVQuadRhsFnB` function type does not have a return value.

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

The `y`, `yB`, and `yQBdot` arguments are all of type `N_Vector`, but they typically all 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 repsect to their `N_Vector` arguments (see §8.1 and §8.2).

The `fQ_dataB` pointer is passed to the user's `fQB` function every time it is called and can be the same as the `f_data` pointer used for the forward problem.

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` is called in step 17 of §7.1), the user may provide, through a call to `CVDenseSetJacFnB` (see §7.2.6), a function of the following type:

`CVDenseJacFnB`

Definition
```
typedef void (*CVDenseJacFnB)(long int nB, DenseMat JB, realtype t,
                              N_Vector y, N_Vector yB, N_Vector fyB,
                              void *jac_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). If the backward problem is the adjoint of the original IVP, then this Jacobian is just the transpose of $J = \partial f/\partial y$ with a change in sign.

Arguments
- `nB` is the backward problem size.
- `J` is the output Jacobian matrix.
- `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 dependent variable vector.
- `fyB` is the current value of the right-hand side of the backward problem.
- `jac_dataB` is a pointer to user data - the same as the `jac_dataB` parameter passed to `CVDenseSetJacDataB`.

tmp1B
tmp2B
tmp3B are pointers to memory allocated for variables of type `N_Vector` which can be used by `CVDenseJacFnB` as temporary storage or work space.

Return value A `CVDenseJacFnB` function type does not have a return value.

Notes A user-supplied dense Jacobian function must load the `nB` by `nB` dense matrix `JB` with an approximation to the Jacobian matrix at the point (`t`,`y`,`yB`), where `y` is the solution of the original IVP at time `t` and `yB` is the solution of the backward problem at the same time. Only nonzero elements need to be loaded into `JB` as this matrix is set to zero before the call to the Jacobian function. The type of `JB` is `DenseMat`. The user is referred to §5.6.3 for details regarding accessing a `DenseMat` object.

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` is called in step 17 of §7.1), the user may provide, through a call to `CVBandSetJacFnB` (see §7.2.6), a function the following type:

`CVBandJacFnB`

Definition
```
typedef void (*CVBandJacFnB)(long int nB,
                              long int mupperB, long int mlowerB,
                              BandMat JB, realtype t, N_Vector y,
                              N_Vector yB, N_Vector fyB, void *jac_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
- `nB` is the backward problem size.
- `mlowerB`
- `mupperB` are the lower and upper half-bandwidth of the Jacobian.
- `JB` is the output Jacobian matrix.
- `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 dependent variable vector.
- `fyB` is the current value of the right-hand side of the backward problem.
- `jac_dataB` is a pointer to user data - the same as the `jac_dataB` parameter passed to `CVBandSetJacDataB`.
- `tmp1B`
- `tmp2B`
- `tmp3B` are pointers to memory allocated for variables of type `N_Vector` which can be used by `CVBandJacFnB` as temporary storage or work space.

Return value A `CVBandJacFnB` function type does not have a return value.

Notes A user-supplied band Jacobian function must load the band matrix `JB` (of type `BandMat`) with the elements of the Jacobian at the point (`t`,`y`,`yB`), where `y` is the solution of the original IVP at time `t` and `yB` is the solution of the backward problem at the same time. Only nonzero elements need to be loaded into `JB` because `JB` is preset to zero before the call to the Jacobian function. More details on the accessor macros provided for a `BandMat` object and on the rest of the arguments passed to a function of type `CVBandJacFnB` are given in §5.6.4.

Jacobian information for the backward problem (SPGMR case)

If an iterative SPGMR linear solver is selected (`CVSpgmrB` is called in step 17 of §7.1), the user may provide, through a call to `CVSpgmrSetJacTimesVecFnB` (see §7.2.6), a function of the following type:

`CVSpgmrJacTimesVecFnB`

Definition
```
typedef int (*CVSpgmrJacTimesVecFnB)(N_Vector vB, N_Vector JvB,
                                      realtype t, N_Vector y,
                                      N_Vector yB, N_Vector fyB,
                                      void *jac_dataB, N_Vector tmpB);
```

Purpose: This function computes the action of the Jacobian on a given vector `vB` for the backward problem (or an approximation to it).

Arguments:
- `vB` is the vector by which the Jacobian must be multiplied to the right.
- `JvB` is the output vector computed.
- `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 dependent variable vector.
- `fyB` is the current value of the right-hand side of the backward problem.
- `jac_dataB` is a pointer to user data - the same as the `jac_dataB` parameter passed to `CVSpgmrSetJacDataB`.
- `tmpB` is a pointer to memory allocated for a vector which can be used for work space.

Return value: The return value of a function of type `CVSpgmrJtimesFnB` 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 result of the product between 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 `CVSpgmrJtimesFn` (see §5.6.5). If the backward problem is the adjoint of $\dot{y} = f(t, y)$, then this function is to compute $-(\partial f/\partial y)^T v_B$.

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. This function must be of type `CVSpgmrPSolveFnB` defined by

`CVSpgmrPrecSolveFnB`

Definition
```
typedef int (*CVSpgmrPrecSolveFnB)(realtype t, N_Vector y, N_Vector yB,
                                    N_Vector fyB, N_Vector rB,
                                    N_Vector zB, realtype gammaB,
                                    realtype deltaB, int lrB,
                                    void *P_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 dependent variable vector.
- `fyB` is the current value of the right-hand side of the backward problem.
- `rB` is the right-hand side vector of the linear system.

- `zB` is the output vector computed.
- `gammaB` is the scalar γ appearing in the Newton matrix.
- `deltaB` is an input tolerance to be used if an iterative method is employed in the solution.
- `lrB` is an input flag indicating whether the preconditioner solve function is to use the left preconditioner (`lr=1`) or the right preconditioner (`lr=2`);
- `P_dataB` is a pointer to user data - the same as the `P_dataB` parameter passed to the function `CVSpgmrSetPrecDataB`.
- `tmpB` is a pointer to memory allocated for a 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).

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 type `CVSpgmrPrecondFnB` defined by

`CVSpgmrPrecSetupFnB`

Definition

```
typedef int (*CVSpgmrPrecSetupFnB)(realtype t, N_Vector y, N_Vector yB,
                                   N_Vector fyB, booleantype jokB,
                                   booleantype *jcurPtrB,
                                   realtype gammaB, void *P_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 `CVSpgmrPrecSetupFnB` 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 dependent variable vector.
- `fyB` is the current value of the right-hand side of the backward problem.
- `jokB` is an input flag indicating whether Jacobian-related data needs to be recomputed.
- `jcurPtrB` is a pointer to an output integer flag which is to be set to `TRUE` if Jacobian data was recomputed, or to `FALSE` if Jacobian data was not recomputed but saved data was still reused.
- `gammaB` is the scalar γ appearing in the Newton matrix.
- `P_dataB` is a pointer to user data - the same as the `P_dataB` parameter passed to `CVSpgmrSetPrecDataB`.
- `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).

7.4 Using CVODES preconditioner modules for the backward problem

7.4.1 Using the banded preconditioner CVBANDPRE

The adjoint module in CVODES offers an interface to the banded preconditioner module CVBANDPRE described in section §5.9.1. This preconditioner provides a band matrix preconditioner based on difference quotients of the backward problem right-hand side function `fB`. It generates a banded approximation to the Jacobian with m_{lB} sub-diagonals and m_{uB} super-diagonals to be used with the Krylov linear solver in CVSPGMR.

In order to use the CVBANDPRE module in the solution of the backward problem, the user need not define any additional functions. First, the user must initialize the CVBANDPRE module by calling

`CVBandPrecAllocB`

Call `flag = CVBandPrecAlloc(cvadj_mem, nB, muB, mlB);`

Description The function `CVBandPrecAllocB` initializes and allocates memory for the CVBANDPRE preconditioner for the backward problem.

Arguments
- `cvadj_mem` (`void *`) pointer to the adjoint memory block returned by `CVadjMalloc`.
- `nB` (`long int`) backward problem dimension.
- `muB` (`long int`) upper half-bandwidth of the backward problem Jacobian approximation.
- `mlB` (`long int`) lower half-bandwidth of the backward problem Jacobian approximation.

Return value If successful, `CVBandPrecAlloc` stores a pointer to the newly created CVBANDPRE memory block. The return value `flag` (of type `int`) is one of:
- `CV_SUCCESS` The call to `CVBandPrecAllocB` was successful.
- `CV_PDATA_NULL` An error occured while trying to create the CVBANDPRE memory block.
- `CV_ADJMEM_NULL` The `cvadj_mem` argument was `NULL`.

To specify the use of the CVSPGMR linear solver module with the CVBANDPRE preconditioner module, make the following call:

`CVBPSpgmrB`

Call `flag = CVBPSpgmrB(cvadj_mem, pretypeB, maxlB);`

Description The function `CVBPSpgmrB` links the CVBANDPRE data to the CVSPGMR linear solver and attaches the latter to the CVODES memory block for the backward problem.

Arguments
- `cvadj_mem` (`void *`) pointer to the adjoint memory block returned by `CVadjMalloc`.
- `pretypeB` (`int`) preconditioning type. Can be one of `PREC_LEFT` or `PREC_RIGHT`.
- `maxlB` (`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `CVSPGMR_MAXL`= 5.

Return value The return value `flag` (of type `int`) is one of:
- `CV_SUCCESS` The CVSPGMR initialization was successful.
- `CVSPGMR_MEM_NULL` The CVODES memory block for the backward problem was not initialized through a previous call to `CVodeCreateB`.
- `CVSPGMR_ILL_INPUT` The preconditioner type `pretypeB` is not valid.
- `CVSPGMR_MEM_FAIL` A memory allocation request failed.
- `CV_PDATA_NULL` The CVBANDPRE preconditioner has not been initialized.
- `CV_ADJMEM_NULL` The `cvadj_mem` argument was `NULL`.

Note that memory allocated by CVBandPrecAllocB is deallocated by the function CVadjFree. For more details on CVBANDPRE see §5.9.1.

7.4.2 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 §5.9.2. This generates a preconditioner that is a block-diagonal matrix with each block being a band matrix and can be used with the Krylov linear solver in CVSPGMR and with the 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.

Usage of CVBBDPRE for the backward problem

The CVBBDPRE module is initialized by calling

CVBBDPrecAllocB

Call
```
flag = CVBBDPrecAllocB(cvadj_mem, NlocalB, mudqB, mldqB,
                       mukeepB, mlkeepB, dqrelyB, glocB, cfnB);
```

Description The function CVBBDPrecAllocB initializes and allocates memory for the CVBBDPRE preconditioner for the backward problem.

Arguments
- cvadj_mem (void *) pointer to the adjoint memory block returned by CVadjMalloc.
- 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 (CVLocalFnB) the C function which computes the approximation $g_B(t, y)$ to the right-hand side of the backward problem.
- cfnB (CVCommFnB) the optional C function which performs all interprocess communication required for the computation of $g_B(t, y)$.

Return value If successful, CVBBDPrecAlloc stores a pointer to the newly created CVBBDPRE memory block. The return value flag (of type int) is one of:
- CV_SUCCESS The call to CVBBDPrecAllocB was successful.
- CV_PDATA_NULL An error occured while trying to create the CVBBDPRE memory block.
- CV_ADJMEM_NULL The cvadj_mem argument was NULL.

To specify the use of the CVSPGMR linear solver module with the CVBBDPRE preconditioner module, make the following call:

CVBBDSpgmrB

Call
```
flag = CVBBDSpgmrB(cvadj_mem, pretypeB, maxlB);
```

Description The function CVBBDSpgmrB links the CVBBDPRE data to the CVSPGMR linear solver and attaches the latter to the CVODES memory block for the backward problem.

Arguments
- `cvadj_mem` (`void *`) pointer to the adjoint memory block returned by `CVadjMalloc`.
- `pretypeB` (`int`) preconditioning type. Can be one of `PREC_LEFT` or `PREC_RIGHT`.
- `maxlB` (`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `CVSPGMR_MAXL`= 5.

Return value The return value `flag` (of type `int`) is one of:
- `CVSPGMR_SUCCESS` The CVSPGMR initialization was successful.
- `CVSPGMR_MEM_NULL` The CVODES memory block for the backward problem was not initialized through a previous call to `CVodeCreateB`.
- `CVSPGMR_ILL_INPUT` The preconditioner type `pretypeB` is not valid.
- `CVSPGMR_MEM_FAIL` A memory allocation request failed.
- `CV_PDATA_NULL` The CVBBDPRE preconditioner has not been initialized.
- `CV_ADJMEM_NULL` The `cvadj_mem` argument was `NULL`.

To reinitialize the CVBBDPRE preconditioner module for the backward problem call the following function:

`CVBBDPrecReInitB`

Call `flag = CVBBDPrecReInitB(cvadj_mem, mudqB, mldqB, dqrelyB, glocB, cfnB);`

Description The function `CVBBDPrecReInitB` reinitializes the CVBBDPRE preconditioner for the backward problem.

Arguments
- `cvadj_mem` (`void *`) pointer to the adjoint memory block returned by `CVadjMalloc`.
- `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.
- `glocB` (`CVLocalFnB`) the C function which computes the approximation $g_B(t, y)$ to the right-hand side of the backward problem.
- `cfnB` (`CVCommFnB`) the optional C function which performs all interprocess communication required for the computation of $g_B(t, y)$.

Return value The return value `flag` (of type `int`) is one of:
- `CV_SUCCESS` The CVSPGMR initialization was successful.
- `CV_ADJMEM_NULL` The `cvadj_mem` argument was `NULL`.

Note that memory allocated by `CVBBDPrecAllocB` is deallocated by the function `CVadjFree`. For more details on CVBBDPRE see §5.9.2.

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 `CVLocalFnB`) which approximates the right-hand side of the backward problem and which is computed locally, and an optional function `cfnB` (of type `CVCommFnB`) which performs all interprocess communication necessary to evaluate this approximate right-hand side (see §5.9.2). The prototypes for these two functions are described below.

`CVLocalFnB`

Definition
```
typedef void (*CVLocalFnB)(long int NlocalB, realtype t,
                           N_Vector y, N_Vector yB, N_Vector gB,
                           void *f_dataB);
```

Purpose This function loads the vector `gB` 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 dependent variable vector.
- `gB` is the output vector.
- `f_dataB` is a pointer to user data - the same as the `f_dataB` parameter passed to `CVodeSetFdataB`.

Return value A `CVLocalFnB` function type does not have a return value.

Notes This routine assumes that all interprocess communication of data needed to calculate `gB` has already been done, and this data is accessible within `f_dataB`.

CVCommFnB

Definition
```
typedef void (*CVCommFnB)(long int NlocalB, realtype t,
                          N_Vector y, N_Vector yB, void *f_dataB);
```

Purpose This function performs 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 dependent variable vector.
- `f_dataB` is a pointer to user data - the same as the `f_dataB` parameter passed to `CVodeSetFdataB`.

Return value A `CVCommFnB` function type does not have a return value.

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

Each call to the `cfnB` 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 `cfnB` = `NULL` to `CVBBDPrecAllocB`.

Chapter 8

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 two provided within SUNDIALS, a serial and an MPI parallel implementations.

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 also 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 8.1 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 8.1: 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 the data array.
`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.
`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 linear solvers, as well as the interfaces to the banded preconditioners 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 linear solver.
`N_VLinearSum`	`N_VLinearSum(a, x, b, y, z);` Performs the operation $z = ax + by$, where a and b are 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 `c`: $z_i = c$, $i = 0, \ldots, n-1$.
`N_VProd`	`N_VProd(x, y, z);` Sets the `N_Vector` `z` to be the component-wise product of the `N_Vector` inputs `x` and `y`: $z_i = x_i y_i$, $i = 0, \ldots, n-1$.
`N_VDiv`	`N_VDiv(x, y, z);` Sets the `N_Vector` `z` to be the component-wise ratio of the `N_Vector` inputs `x` and `y`: $z_i = x_i / y_i$, $i = 0, \ldots, n-1$. The y_i may not be tested for 0 values. It should only be called with an `x` that is guaranteed to have all nonzero components.

continued on next page

continued from last page

Name	Usage and Description
`N_VScale`	`N_VScale(c, x, z);` Scales the `N_Vector` `x` by the scalar `c` and returns the result in `z`: $z_i = cx_i$, $i = 0, \ldots, n-1$.
`N_VAbs`	`N_VAbs(x, y);` Sets the components of the `N_Vector` `y` to be the absolute values of the components of the `N_Vector` `x`: $y_i = \lvert x_i \rvert$, $i = 0, \ldots, n-1$.
`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 scalar `b` to all components of `x` and returns the result in the `N_Vector` `z`: $z_i = x_i + b$, $i = 0, \ldots, n-1$.
`N_VDotProd`	`d = N_VDotProd(x, y);` Returns the value of the ordinary dot product of `x` and `y`: $d = \sum_{i=0}^{n-1} x_i y_i$.
`N_VMaxNorm`	`m = N_VMaxNorm(x);` Returns the maximum norm of the `N_Vector` `x`: $m = \max_i \lvert x_i \rvert$.
`N_VWrmsNorm`	`m = N_VWrmsNorm(x, w)` Returns the weighted root-mean-square norm of the `N_Vector` `x` with 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 weight vector `w` built using only the elements of `x` corresponding to nonzero elements of the `N_Vector` `id`: $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i \operatorname{sign}(id_i))^2 \right) / n}$.
`N_VMin`	`m = N_VMin(x);` Returns the smallest element of the `N_Vector` `x`: $m = \min_i x_i$.
`N_VWL2Norm`	`m = N_VWL2Norm(x, w);` Returns the weighted Euclidean ℓ_2 norm of the `N_Vector` `x` with 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`: $m = \sum_{i=0}^{n-1} \lvert x_i \rvert$.

continued on next page

continued from last page	
Name	**Usage and Description**
N_VCompare	`N_VCompare(c, x, z);` Compares the components of the `N_Vector` `x` to the scalar `c` and returns an `N_Vector` `z` such that: $z_i = 1.0$ if $\|x_i\| \geq c$ 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 `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 `FALSE` if any element failed the constraint test, `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 `sundialstypes.h`) is returned.

8.1 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 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 8.1 and provides the following user-callable routines:

- `N_VNew_Serial`

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

  ```
  N_Vector N_VNew_Serial(long int vec_length);
  ```

- `N_VNewEmpty_Serial`

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

  ```
  N_Vector N_VNewEmpty_Serial(long int vec_length);
  ```

- `N_VMake_Serial`

  This function creates and allocates memory for a serial vector with user-provided data array.

  ```
  N_Vector N_VMake_Serial(long int vec_length, realtype *v_data);
  ```

- `N_VNewVectorArray_Serial`

  This function creates an array of `count` serial vectors.

  ```
  N_Vector *N_VNewVectorArray_Serial(int count, long int vec_length);
  ```

- `N_VNewVectorArrayEmpty_Serial`

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

  ```
  N_Vector *N_VNewVectorArrayEmpty_Serial(int count, long int vec_length);
  ```

- `N_VDestroyVectorArray_Serial`

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

- ! The NVECTOR_SERIAL constructor functions N_VNewEmpty_Serial, N_VMake_Serial, and N_VNewVectorArrayEmpty_Serial set the field *own_data* = FALSE. The functions 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.

8.2 The NVECTOR_PARALLEL implementation

The parallel implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_PARALLEL, defines the *content* field of N_Vector to be a structure containing the global and local lengths of the vector, a pointer to the beginning of a contiguous local data array, an MPI communicator, an a boolean flag *own_data* indicating ownership of the data array *data*.

```
struct _N_VectorContent_Parallel {
  long int local_length;
  long int global_length;
  booleantype own_data;
  realtype *data;
  MPI_Comm comm;
};
```

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

- NV_CONTENT_P

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

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

  Implementation:

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

- NV_OWN_DATA_P, NV_DATA_P, NV_LOCLENGTH_P, NV_GLOBLENGTH_P

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

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

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

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

  Implementation:

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

- `NV_COMM_P`

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

  Implementation:

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

- `NV_Ith_P`

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

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

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

  Implementation:

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

The NVECTOR_PARALLEL module defines parallel implementations of all vector operations listed in Table 8.1 and provides the following 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_VNewVectorArray_Parallel`

  This function creates an array of `count` parallel vectors.

```
N_Vector *N_VNewVectorArray_Parallel(int count,
                                     MPI_Comm comm,
                                     long int local_length,
                                     long int global_length);
```

- `N_VNewVectorArrayEmpty_Parallel`

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

```
N_Vector *N_VNewVectorArrayEmpty_Parallel(int count,
                                          MPI_Comm comm,
                                          long int local_length,
                                          long int global_length);
```

- `N_VDestroyVectorArray_Parallel`

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

- ! The NVECTOR_PARALLEL constructor functions `N_VNewEmpty_Parallel`, `N_VMake_Parallel`, and `N_VNewVectorArrayEmpty_Parallel` set the field *own_data* = `FALSE`. The functions `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 responsability to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

8.3 NVECTOR functions used by CVODES

In Table 8.2 below, we list the vector functions in the NVECTOR module 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 seven columns show function usage within each of the four CVODES linear solvers, the CVBANDPRE and CVBBDPRE preconditioner modules, and the CVODEA adjoint sensitivity module.

There is one subtlety in the CVSPGMR column hidden by the table. The dot product function `N_VDotProd` is called both within the implementation file `cvspgmr.c` for the CVSPGMR solver and within the implementation files `spgmr.c` and `iterative.c` for the generic SPGMR solver upon which the CVSPGMR solver is implemented. Also, although `N_VDiv` and `N_VProd` are not called within the implementation file `cvspgmr.c`, they are called within the implementation file `spgmr.c` and so are required by the CVSPGMR solver module. 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.

Table 8.2: List of vector functions usage by CVODES code modules

	CVODES	CVDENSE	CVBAND	CVDIAG	CVSPGMR	CVBANDPRE	CVBBDPRE	CVODEA
N_VClone	✓			✓	✓			✓
N_VDestroy	✓			✓	✓			✓
N_VSpace	✓							
N_VGetArrayPointer		✓	✓			✓	✓	
N_VSetArrayPointer		✓						
N_VLinearSum	✓	✓		✓	✓			✓
N_VConst	✓				✓			
N_VProd	✓			✓	✓			
N_VDiv	✓			✓	✓			
N_VScale	✓	✓	✓	✓	✓	✓	✓	✓
N_VAbs	✓							
N_VInv	✓			✓				
N_VAddConst	✓			✓				
N_VDotProd					✓			
N_VMaxNorm	✓							
N_VWrmsNorm	✓	✓	✓		✓	✓	✓	
N_VMin	✓							
N_VCompare				✓				
N_VInvTest				✓				

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 8.1 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 five kernels.

Chapter 9

Providing Alternate Linear Solver Modules

The central CVODES module interfaces with the linear solver module to be used by way of calls to four routines. These are denoted here by `linit`, `lsetup`, `lsolve`, and `lfree`. Briefly, their purposes are as follows:

- `linit`: initialize and allocate memory specific to the linear solver;
- `lsetup`: preprocess and evaluate the Jacobian or preconditioner;
- `lsolve`: solve the linear system;
- `lfree`: free the linear solver memory.

A linear solver module must also provide a user-callable specification routine (like those described in §5.5.2) which will attach the above four routines to the main CVODES memory block. The return value of the specification routine should be: `CV*_SUCCESS` = 0 if the routine was successful, `CV*_MEM_FAIL` = -4 if a memory allocation failed, `CV*_ILL_INPUT` = -3 if some input was illegal or the NVECTOR implementation is not compatible, or `CV*_MEM_NULL` = -1 if the pointer to the main CVODES memory block is `NULL`.

These four routines that 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 argument list for each of these routines. Note that the argument list of each routine includes a pointer to the main CVODES memory block, by which the routine can access various data related to the CVODES solution. The contents of this memory block are given in the file `cvodes.h` (but not reproduced here, for the sake of space).

Initialization routine. The type definition of `linit` is

`linit`

Definition `int (*linit)(CVodeMem cv_mem);`

Purpose The purpose of `linit` is to complete linear solver-specific initializations, such as counters and statistics.

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 -1 otherwise.

Notes If an error does occur, an appropriate message should be sent to `cv_mem->cv_errfp`.

Setup routine. 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`. It may recompute Jacobian-related data if it is deemed necessary.

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:
  - `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).
  - `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 routine indicated that its Jacobian-related data is not current, or (b) during the previous Newton corrector iteration, the linear solver's solve routine failed in a recoverable manner and the linear solver's setup routine indicated that its Jacobian-related data is not current.
  - `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`, i.e. $f(t_n, y_{pred})$.
- `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 reevaluation 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` routine should return 0 if successful, a positive value for a recoverable error, and a negative value for an unrecoverable error.

Solve routine. 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 routine `lsolve` must solve the linear equation $Mx = b$, where M is some approximation to $I - \gamma J$, $J = (\partial f/\partial y)(t_n, y_{cur})$ and the right-hand side vector b is input.

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 reciprocals of the W_i of (3.7).

`ycur` is a vector that contains the solver's current approximation to $y(t_n)$.

`fcur` is a vector that contains $f(t_n, y_{cur})$.

Return value `lsolve` returns a positive value for a recoverable error and a negative value for an unrecoverable error. Success is indicated by a 0 return value.

Memory deallocation routine. The type definition of `lfree` is

`lfree`

Definition `void (*lfree)(CVodeMem cv_mem);`

Purpose The routine `lfree` should free up any memory allocated by the linear solver.

Arguments The argument `cv_mem` is the CVODES memory pointer of type `CVodeMem`.

Return value This routine has no return value.

Notes This routine is called once a problem has been completed and the linear solver is no longer needed.

Chapter 10

Generic Linear Solvers in SUNDIALS

In this chapter, we describe three generic linear solver code modules that are included in CVODES, but which are of potential use as generic packages in themselves, either in conjunction with the use of CVODES or separately. These modules are:

- The DENSE matrix package, which includes the matrix type `DenseMat`, macros and functions for `DenseMat` matrices, and functions for small dense matrices treated as simple array types.
- The BAND matrix package, which includes the matrix type `BandMat`, macros and functions for `BandMat` matrices, and functions for small band matrices treated as simple array types.
- The SPGMR package, which includes a solver for the scaled preconditioned GMRES method.

For the sake of space, the functions for `DenseMat` and `BandMat` matrices and the functions in SPGMR are only summarized briefly, since they are less likely to be of direct use in connection with CVODES. The functions for small dense matrices are fully described, because we expect that they will be useful in the implementation of preconditioners used with the combination of CVODES and the CVSPGMR solver.

10.1 The DENSE module

10.1.1 Type `DenseMat`

The type `DenseMat` is defined to be a pointer to a structure with a size and a data field:

```
typedef struct {
  long int size;
  realtype  **data;
} *DenseMat;
```

The *size* field indicates the number of columns (which is the same as the number of rows) of a dense matrix, while the *data* field is a two dimensional array used for component storage. The elements of a dense matrix are stored columnwise (i.e columns are stored one on top of the other in memory). If `A` is of type `DenseMat`, then the (`i`,`j`)-th element of `A` (with $0 \leq$ `i`, `j` $\leq$ `size`-1) is given by the expression `(A->data)[j][i]` or by the expression `(A->data)[0][j*size+i]`. 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` macro. Users should use these macros whenever possible.

10.1.2 Accessor Macros

The following two macros are defined by the DENSE module to provide access to data in the `DenseMat` 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 $N \times N$ `DenseMat A`, $0 \le$ `i`, `j` $\le N-1$.

- `DENSE_COL`

  Usage : `col_j = DENSE_COL(A,j);`

  `DENSE_COL` references the `j`-th column of the $N \times N$ `DenseMat A`, $0 \le$ `j` $\le N-1$. 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 $N-1$. The (`i`, `j`)-th element of `A` is referenced by `col_j[i]`.

10.1.3 Functions

The following functions for `DenseMat` matrices are available in the DENSE package. For full details, see the header file `dense.h`.

- `DenseAllocMat`: allocation of a `DenseMat` matrix;
- `DenseAllocPiv`: allocation of a pivot array for use with `DenseFactor`/`DenseBacksolve`;
- `DenseFactor`: LU factorization with partial pivoting;
- `DenseBacksolve`: solution of $Ax = b$ using LU factorization;
- `DenseZero`: load a matrix with zeros;
- `DenseCopy`: copy one matrix to another;
- `DenseScale`: scale a matrix by a scalar;
- `DenseAddI`: increment a matrix by the identity matrix;
- `DenseFreeMat`: free memory for a `DenseMat` matrix;
- `DenseFreePiv`: free memory for a pivot array;
- `DensePrint`: print a `DenseMat` matrix to standard output.

10.1.4 Small Dense Matrix Functions

The following functions for small dense matrices are available in the DENSE package:

- `denalloc`

  `denalloc(n)` allocates storage for an `n` by `n` dense matrix. It returns a pointer to the newly allocated storage if successful. If the memory request cannot be satisfied, then `denalloc` returns `NULL`. The underlying type of the dense matrix returned is `realtype**`. If we allocate a dense matrix `realtype** a` by `a = denalloc(n)`, then `a[j][i]` references the (`i`,`j`)-th element of the matrix `a`, $0 \le$ `i`, `j` $\le$ `n`-1, 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 `n`^{2} contiguous locations which contain the elements of `a`.

- `denallocpiv`

  `denallocpiv(n)` allocates an array of `n` integers. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

- `gefa`

  `gefa(a,n,p)` factors the `n` by `n` dense matrix `a`. 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` $= 0, 1, \ldots,$`n`-1.
  2. If the unique LU factorization of `a` is given by $Pa = LU$, where P is a permutation matrix, L is a lower triangular matrix with all 1's on the diagonal, and U is an upper triangular matrix, then the upper triangular part of `a` (including its diagonal) contains U and the strictly lower triangular part of `a` contains the multipliers, $I - L$.

     `gefa` returns 0 if successful. Otherwise it encountered a zero diagonal element during the factorization. In this case it returns the column index (numbered from one) at which it encountered the zero.

- `gesl`

  `gesl(a,n,p,b)` solves the `n` by `n` linear system $ax = b$. It assumes that `a` has been LU-factored and the pivot array `p` has been set by a successful call to `gefa(a,n,p)`. The solution x is written into the `b` array.

- `denzero`

  `denzero(a,n)` sets all the elements of the `n` by `n` dense matrix `a` to be 0.0;

- `dencopy`

  `dencopy(a,b,n)` copies the `n` by `n` dense matrix `a` into the `n` by `n` dense matrix `b`;

- `denscale`

  `denscale(c,a,n)` scales every element in the `n` by `n` dense matrix `a` by `c`;

- `denaddI`

  `denaddI(a,n)` increments the `n` by `n` dense matrix `a` by the identity matrix;

- `denfreepiv`

  `denfreepiv(p)` frees the pivot array `p` allocated by `denallocpiv`;

- `denfree`

  `denfree(a)` frees the dense matrix `a` allocated by `denalloc`;

- `denprint`

  `denprint(a,n)` prints the `n` by `n` dense matrix `a` to standard output as it would normally appear on paper. It is intended as a debugging tool with small values of `n`. The elements are printed using the `%g` option. A blank line is printed before and after the matrix.

10.2 The BAND module

10.2.1 Type BandMat

The type `BandMat` is the type of a large band matrix A (possibly distributed). It is defined to be a pointer to a structure defined by:

```
typedef struct {
  long int size;
  long int mu, ml, smu;
  realtype **data;
} *BandMat;
```

The fields in the above structure are:

- *size* is the number of columns (which is the same as the number of rows);
- *mu* is the upper half-bandwidth, $0 \leq mu \leq size{-}1$;
- *ml* is the lower half-bandwidth, $0 \leq ml \leq size{-}1$;
- *smu* is the storage upper half-bandwidth, $mu \leq smu \leq size{-}1$. The `BandFactor` routine writes the LU factors into the storage for A. The upper triangular factor U, however, may have an upper half-bandwidth as big as min(*size*−1,*mu*+*ml*) because of partial pivoting. The *smu* field holds the upper half-bandwidth allocated for A.
- *data* is a two dimensional array used for component storage. The elements of a band matrix of type `BandMat` 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.

  If we number rows and columns in the band matrix starting from 0, then

  – *data[0]* is a pointer to (*smu*+*ml*+1)**size* contiguous locations which hold the elements within the band of A

  – *data[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 *smu*−*mu* (to access the uppermost element within the band in the j-th column) to *smu*+*ml* (to access the lowest element within the band in the j-th column). Indices from 0 to *smu*−*mu*−1 give access to extra storage elements required by `BandFactor`.

  – *data[j][i-j+smu]* is the (i, j)-th element, $j{-}mu \leq i \leq j{+}ml$.

The macros below allow a user to 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 into the `j`-th column of elements can be obtained via the `BAND_COL` macro. Users should use these macros whenever possible.

See Figure 10.1 for a diagram of the `BandMat` type.

10.2.2 Accessor Macros

The following three macros are defined by the BAND module to provide access to data in the `BandMat` 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)`.

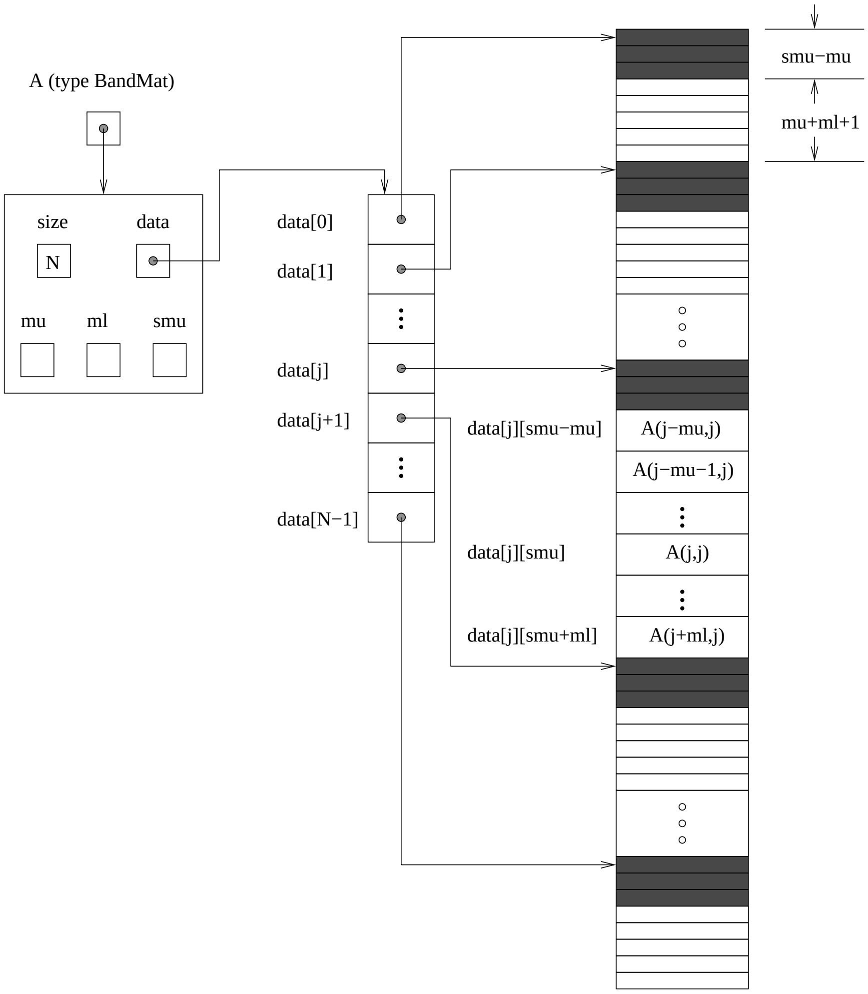


Figure 10.1: Diagram of the storage for a band matrix of type `BandMat`. Here `A` is an $N \times N$ band matrix of type `BandMat` with upper and lower half-bandwidths `mu` and `ml`, respectively. The rows and columns of `A` are numbered from 0 to $N-1$ and the (i,j)-th element of `A` is denoted `A(i,j)`. The greyed out areas of the underlying component storage are used by the `BandFactor` and `BandBacksolve` routines.

- 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).

10.2.3 Functions

The following functions for BandMat matrices are available in the BAND package. For full details, see the header file band.h.

- BandAllocMat: allocation of a BandMat matrix;
- BandAllocPiv: allocation of a pivot array for use with BandFactor/BandBacksolve;
- BandFactor: LU factorization with partial pivoting;
- BandBacksolve: solution of $Ax = b$ using LU factorization;
- BandZero: load a matrix with zeros;
- BandCopy: copy one matrix to another;
- BandScale: scale a matrix by a scalar;
- BandAddI: increment a matrix by the identity matrix;
- BandFreeMat: free memory for a BandMat matrix;
- BandFreePiv: free memory for a pivot array;
- BandPrint: print a BandMat matrix to standard output.

10.3 The SPGMR module

The SPGMR package, in the files spgmr.h and spgmr.c, includes an implementation of the scaled preconditioned GMRES method. A separate code module, iterative.h and iterative.c, contains auxiliary functions that support SPGMR, and also other Krylov solvers to be added later. For full details, including usage instructions, see the files spgmr.h and iterative.h.

Functions. 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 iterative.h and iterative.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`.

Chapter 11

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.

11.1 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_SS	1	Scalar relative tolerance, scalar absolute tolerance.
CV_SV	2	Scalar relative tolerance, vector absolute tolerance.
CV_EE	2	Estimated relative tolerance and absolute tolerance for sensitivity variables.
CV_NORMAL	1	Solver returns at specified output time.
CV_ONE_STEP	2	Solver returns after each successful step.
CV_NORMAL_TSTOP	3	Solver returns at specified output time, but does not proceed past the specified stopping time.
CV_ONE_STEP_TSTOP	4	Solver returns after each successful step, but does not proceed past the specified stopping time.
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.

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.

11.2 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_MEM_NULL	-1	The cvode_mem argument was NULL.
CV_ILL_INPUT	-2	One of the function inputs is illegal.
CV_NO_MALLOC	-3	The CVODE memory was not allocated by a call to CVodeMalloc.
CV_TOO_MUCH_WORK	-4	The solver took mxstep internal steps but could not reach tout.
CV_TOO_MUCH_ACC	-5	The solver could not satisfy the accuracy demanded by the user for some internal step.
CV_ERR_FAILURE	-6	Error test failures occurred too many times during one internal time step or minimum step size was reached.
CV_CONV_FAILURE	-7	Convergence test failures occurred too many times during one internal time step or minimum step size was reached.
CV_LINIT_FAIL	-8	The linear solver's initialization function failed.
CV_LSETUP_FAIL	-9	The linear solver's setup function failed in an unrecoverable manner.
CV_LSOLVE_FAIL	-10	The linear solver's solve function failed in an unrecoverable manner.
CV_MEM_FAIL	-11	A memory allocation failed.
CV_RTFUNC_NULL	-12	The user-supplied root function is NULL.
CV_NO_SLDET	-13	The stability limit detection algorithm was not activated.
CV_BAD_K	-14	The derivative order k is larger than the order used.
CV_BAD_T	-15	The time t s outside the last step taken.
CV_BAD_DKY	-16	The output derivative vector is NULL.
CV_PDATA_NULL	-17	The preconditioner module has not been initialized.
CV_BAD_IS	-18	The sensitivity index is larger than the number of sensitivities computed.
CV_NO_QUAD	-20	Quadrature integration was not activated.
CV_NO_SENS	-21	Forward sensitivity integration was not activated.

CVODEA adjoint solver module

CV_ADJMEM_NULL	-101	The cvadj_mem argument was NULL.
CV_BAD_TB0	-103	The final time for the adjoint problem is outside the interval over which the forward problem was solved.
CV_BCKMEM_NULL	-104	The cvodes memory for the backward problem was not created.
CV_REIFWD_FAIL	-105	Reinitialization of the forward problem failed at the first checkpoint.
CV_FWD_FAIL	-106	An error occured during the integration of the forward problem.
CV_BAD_ITASK	-107	Wrong task for backward integration.
CV_BAD_TBOUT	-108	The desired output time is outside the interval over which the forward problem was solved.
CV_GETY_BADT	-109	Wrong time in Hermite interpolation function.

CVDENSE linear solver module

CVDENSE_SUCCESS	0	Successful function return.
CVDENSE_MEM_NULL	-1	The cvode_mem argument was NULL.
CVDENSE_LMEM_NULL	-2	The CVDENSE linear solver has not been initialized.

CVDENSE_ILL_INPUT	-3	The CVDENSE solver is not compatible with the current NVECTOR module.
CVDENSE_MEM_FAIL	-4	A memory allocation request failed.

CVBAND linear solver module

CVBAND_SUCCESS	0	Successful function return.
CVBAND_MEM_NULL	-1	The cvode_mem argument was NULL.
CVBAND_LMEM_NULL	-2	The CVBAND linear solver has not been initialized.
CVBAND_ILL_INPUT	-3	The CVBAND solver is not compatible with the current NVECTOR module.
CVBAND_MEM_FAIL	-4	A memory allocation request failed.

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.

CVSPGMR linear solver module

CVSPGMR_SUCCESS	0	Successful function return.
CVSPGMR_MEM_NULL	-1	The cvode_mem argument was NULL.
CVSPGMR_LMEM_NULL	-2	The CVSPGMR linear solver has not been initialized.
CVSPGMR_ILL_INPUT	-3	The CVSPGMR solver is not compatible with the current NVECTOR module.
CVSPGMR_MEM_FAIL	-4	A memory allocation request failed.

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_MEM_NULL	-1	The SPGMR memory is NULL
SPGMR_ATIMES_FAIL	-2	The Jacobian tims vector function failed.
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.

Bibliography

[1] 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.

[2] P. N. Brown and A. C. Hindmarsh. Reduced Storage Matrix Methods in Stiff ODE Systems. *J. Appl. Math. & Comp.*, 31:49–91, 1989.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] S. D. Cohen and A. C. Hindmarsh. CVODE User Guide. Technical Report UCRL-MA-118618, LLNL, September 1994.

[9] S. D. Cohen and A. C. Hindmarsh. CVODE, a Stiff/Nonstiff ODE Solver in C. *Computers in Physics*, 10(2):138–143, 1996.

[10] 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.

[11] K. L. Hiebert and L. F. Shampine. Implicitly Defined Output Points for Solutions of ODEs. Technical Report SAND80-0180, Sandia National Laboratories, February 1980.

[12] 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.

[13] A. C. Hindmarsh. Avoiding BDF Stability Barriers in the MOL Solution of Advection-Dominated Problems. *Appl. Num. Math.*, 17:311–318, 1995.

[14] A. C. Hindmarsh. The PVODE and IDA Algorithms. Technical Report UCRL-ID-141558, LLNL, December 2000.

[15] 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.*, (submitted), 2004.

[16] A. C. Hindmarsh and R. Serban. Example Programs for CVODE v2.2.0. Technical report, LLNL, 2004. UCRL-SM-208110.

[17] A. C. Hindmarsh and R. Serban. Example Programs for CVODES v2.1.0. Technical Report UCRL-SM-208115, LLNL, 2004.

[18] A. C. Hindmarsh and R. Serban. User Documentation for CVODE v2.2.0. Technical Report UCRL-SM-208108, LLNL, 2004.

[19] 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.

[20] 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.

[21] 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.

[22] T. Maly and L. R. Petzold. Numerical Methods and Software for Sensitivity Analysis of Differential-Algebraic Systems. *Applied Numerical Mathematics*, 20:57–79, 1997.

[23] 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.

Index